Automatic Task and Data Mapping in Shared Memory Architectures

vorgelegt von

Dipl.-Ing.

Matthias Diener

geb. in Berlin

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Sebastian Möller
Stellv. Vorsitzender:	Prof. Dr. Rodolfo Azevedo
Gutachter:	Prof. Dr. Hans-Ulrich Heiß
Gutachter:	Prof. Dr. Jan Richling
Gutachter:	Prof. Dr. Antonio Carlos Schneider Beck Filho
Gutachter:	Prof. Dr. Philippe O. A. Navaux

Tag der wissenschaftlichen Aussprache: 28. Oktober 2015

Berlin 2015

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisors, who have continuously guided and motivated me throughout my research. This thesis would not have been possible without their support.

My thanks also go to all my colleagues and collaborators at the GPPD and KBS research groups, for their comments, discussions and support for my research.

Finally, I am deeply grateful for the encouragement and support of my family during the PhD, as well as in my life in general.

ABSTRACT

Reducing the cost of memory accesses, both in terms of performance and energy consumption, is a major challenge in shared-memory architectures. Modern systems have deep and complex memory hierarchies with multiple cache levels and memory controllers, leading to a Non-Uniform Memory Access (NUMA) behavior. In such systems, there are two ways to improve the memory affinity: First, by mapping tasks that share data (communicate) to cores with a shared cache, cache usage and communication performance are improved. Second, by mapping memory pages to memory controllers that perform the most accesses to them and are not overloaded, the average cost of accesses is reduced. We call these two techniques task mapping and data mapping, respectively. For optimal results, task and data mapping need to be performed in an integrated way. Previous work in this area performs the mapping only separately, which limits the gains that can be achieved. Furthermore, most previous mechanisms require expensive operations, such as communication or memory access traces, to perform the mapping, require changes to the hardware or to the parallel application, or use a simple static mapping. These mechanisms can not be considered generic solutions for the mapping problem. In this thesis, we make two contributions to the mapping problem. First, we introduce a set of metrics and a methodology to analyze parallel applications in order to determine their suitability for an improved mapping and to evaluate the possible gains that can be achieved using an optimized mapping. Second, we propose two automatic mechanisms that perform task mapping and combined task/data mapping, respectively, during the execution of a parallel application. These mechanisms work on the operating system level and require no changes to the hardware, the applications themselves or their runtime libraries. An extensive evaluation with parallel applications from multiple benchmark suites as well as real scientific applications shows substantial performance and energy efficiency improvements that are significantly higher than simple mechanisms and previous work, while maintaining a low overhead.

Keywords: Task mapping. Data mapping. Shared memory. Multicore. NUMA.

Mapeamento automático de processos e dados em arquiteturas de memória compartilhada

RESUMO

Arquiteturas paralelas modernas têm hierarquias de memória complexas, que consistem de vários níveis de memórias cache privadas e compartilhadas, bem como Non-Uniform Memory Access (NUMA) devido a múltiplos controladores de memória por sistema. Um dos grandes desafios dessas arquiteturas é melhorar a localidade e o balanceamento de acessos à memória de tal forma que a latência média de acesso à memória é reduzida. Dessa forma, o desempenho e a eficiência energética de aplicações paralelas podem ser melhorados. Os acessos podem ser melhorados de duas maneiras: (1) processos que acessam dados compartilhados (comunicação entre processos) podem ser alocados em unidades de execução próximas na hierarquia de memória, a fim de melhorar o uso das caches. Esta técnica é chamada de mapeamento de processos. (2) Mapear as páginas de memória que cada processo acessa ao nó NUMA que ele está sendo executado, assim, pode-se reduzir o número de acessos a memórias remotas em arquiteturas NUMA. Essa técnica é conhecida como mapeamento de dados. Para melhores resultados, os mapeamentos de processos e dados precisam ser realizados de forma integrada. Trabalhos anteriores nesta área executam os mapeamentos separadamente, o que limita os ganhos que podem ser alcançados. Além disso, a maioria dos mecanismos anteriores exigem operações caras, como traços de acessos à memória, para realizar o mapeamento, além de exigirem mudanças no hardware ou na aplicação paralela. Estes mecanismos não podem ser considerados soluções genéricas para o problema de mapeamento. Nesta tese, fazemos duas contribuições principais para o problema de mapeamento. Em primeiro lugar, nós introduzimos um conjunto de métricas e uma metodologia para analisar aplicações paralelas, a fim de determinar a sua adequação para um melhor mapeamento e avaliar os possíveis ganhos que podem ser alcançados através desse mapeamento otimizado. Em segundo lugar, propomos um mecanismo que executa o mapeamento de processos e dados online. Este mecanismo funciona no nível do sistema operacional e não requer alterações no hardware, os códigos fonte ou bibliotecas. Uma extensa avaliação com múltiplos conjuntos de carga de trabalho paralelos mostram consideráveis melhorias em desempenho e eficiência energética.

Palavras-chave: Mapeamento de processos. Mapeamento de dados. Memória compartilhada. Multicore. NUMA.

Automatisches Mapping von Tasks und Daten in Shared Memory Architekturen

ZUSAMMENFASSUNG

Moderne parallele Architekturen haben komplexe Speicherhierarchien, die aus mehreren Ebenen von privaten und gemeinsam genutzten Caches, sowie einem Non-Uniform Memory Access (NUMA) Verhalten aufgrund mehrerer Speichercontroller bestehen. Eine der größten Herausforderungen in diesen Architekturen ist die Verbesserung der Lokalität und Balance von Speicherzugriffen, so dass die Latenz der Speicherzugriffe reduziert wird, da dies die Performance und Energieeffizienz von parallelen Anwendungen verbessern kann. Zwei Typen von Verbesserungen existieren: (1) Tasks die auf gemeinsame Daten zugreifen, sollen nah beieinander in der Speicherhierarchie ausgeführt werden, um die Nutzung der Caches zu verbessern. Wir bezeichnen diese Technik als Task Mapping. (2) Speicherseiten sollen den Speichercontrollern zugeordnet werden, von denen die meisten Zugriffe kommen und die nicht überlastet sind, um die durchschnittliche Speicherzugriffszeit zu reduzieren. Wir nennen diese Technik Data Mapping. Für optimale Ergebnisse müssen Task und Data Mapping in integrierter Form durchgeführt werden. Frühere Arbeiten in diesem Bereich führt das Mapping nur getrennt aus, wodurch die Vorteile, die erzielt werden können, begrenzt werden. Außerdem erfordern die meisten früheren Mechanismen teure Operationen um das Mapping durchzuführen, wie etwa Traces der Kommunikation oder von Speicherzugriffen. Andere Arbeiten erfordern Änderungen an der Hardware oder an der parallelen Anwendung, oder erzeugen nur ein einfaches statisches Mapping. Diese Mechanismen können nicht als generische Lösungen für das Mapping-Problem betrachtet werden. In dieser Arbeit präsentieren wir zwei Beiträge für das Mapping-Problem. Zuerst führen wir eine Reihe von Metriken und eine Methodologie zur Bestimmung der Eignung von parallelen Anwendungen für die verschiedenen Arten von Mapping. Das Ziel ist es, die möglichen Gewinne, die durch eine optimiertes Mapping erreicht werden können, vorherzusagen. Zweitens stellen wir zwei Mechanismen vor, die Task und Data Mapping automatisch während der Laufzeit der parallelen Applikation ausführen. Diese Mechanismen funktionieren auf der Betriebssystemebene und erfordern keine Änderungen an der Hardware, den Anwendungen oder deren Laufzeit-Bibliotheken. Eine umfangreiche Auswertung mit einer großen Auswahl paralleler Applikationen zeigen signifikante Verbesserungen der Performance und der Energieeffizienz.

Keywords: Task Mapping. Data Mapping. Shared Memory. Multicore. NUMA.

LIST OF FIGURES

Figure 1.1 Comparison of a shared memory architecture and a cluster system............ 21
Figure 1.2 Scaling of the Intel Xeon processor between 1998 and 2014........................22
Figure 1.3 The impact of memory accesses on performance and energy consumption. .23
Figure 1.4 Comparison of UMA and NUMA architectures...25
Figure 1.5 Example NUMA architecture with 4 NUMA nodes. 27
Figure 1.6 Results of the producer/consumer benchmark. 28
Figure 1.7 Why task mapping is required for data mapping. 29
Figure 1.8 Classification of the memory access behavior of parallel applications. 30
Figure 1.9 Overview of the chapters of the thesis..34

Figure 3.1 Explicit and implicit communication between two tasks.............................46
Figure 3.2 Comparison between true and false communication...................................47
Figure 3.3 Representations of undirected communication behavior..............................49
Figure 3.4 Communication matrices of the accurate and relaxed definitions. 52
Figure 3.5 Common types of communication patterns. 54
Figure 3.5 Common types of communication patterns (ctd.). 55
Figure 3.6 Locality and balance of communication. 56
Figure 3.7 Communication balance. 57
Figure 3.8 Comparison of the structure of communication. 59
Figure 3.9 Inputs for the task mapping example. 64
Figure 3.10 Output of the task mapping example with various algorithms..................65
Figure 3.11 Communication matrices of the NAS-OMP benchmarks...........................66
Figure 3.12 Communication characteristics of the NAS-OMP benchmarks. 67
Figure 3.13 Communication matrices of the PARSEC benchmarks. 68
Figure 3.14 Communication characteristics of the PARSEC benchmarks. 69
Figure 3.15 Communication matrices of the NAS-MPI benchmarks. 70
Figure 3.16 Communication characteristics of the NAS-MPI benchmarks. 70
Figure 3.17 Communication matrices of various phases of the HPCC benchmark. 71
Figure 3.18 Communication characteristics of each phase of the HPCC benchmark. ...72
Figure 3.19 Communication matrices of the NAS-MZ benchmarks..............................72
Figure 3.20 Communication characteristics of the NAS-MZ benchmarks. 72
Figure 3.21 Number of communication phases per second execution time C_D. 73
Figure 3.22 Dynamic communication behavior of the SP-OMP benchmark. 73
Figure 3.23 Dynamic communication behavior of the Ferret benchmark. 74

Figure 4.1 Memory access exclusivity for an application that accesses two pages.........78
Figure 4.2 Memory access balance for two applications and four NUMA nodes. 79
Figure 4.3 Matrix multiplication code. 79
Figure 4.4 Example of data mapping policy behavior. 91
Figure 4.5 Application exclusivity E_{App} for 4 NUMA nodes and different page sizes. ..92
Figure 4.6 Application exclusivity E_{App} for 64 NUMA nodes and different page sizes..93
Figure 4.7 Page and memory access balance of the first-touch policy. 94
Figure 4.8 Page balance B_{Pages} of the data mapping policies. 95
Figure 4.9 Memory access balance B_{Acc} of the data mapping policies. 95
Figure 4.10 Comparing memory access balance between the main mapping policies. ...96
Figure 4.11 Locality of the first-touch page mapping. 96
Figure 4.12 Comparing memory access locality between the main mapping policies. ...97
Figure 4.13 Number of page migrations. 98

Figure 4.14 Dynamic page usage behavior of SP-OMP. 98
Figure 4.15 Dynamic page usage behavior of Vips. 99

Figure 5.1 Performance and energy consumption of SP-OMP in SiNUCA. 105
Figure 5.2 Performance improvements of task mapping policies on *Xeon*. 109
Figure 5.3 Performance improvements of data mapping policies on *Itanium*. 112
Figure 5.4 Performance improvements of data mapping policies on *Xeon*.................. 112
Figure 5.5 Performance improvements of data mapping policies on *Opteron*. 113

Figure 6.1 The communication detection mechanism. 124
Figure 6.2 Example of the operation of CDSM. 125
Figure 6.3 Communication detection accuracy of CDSM. 129
Figure 6.4 Dynamic detection behavior of the SP-OMP benchmark. 130
Figure 6.5 Performance results of CDSM. 131
Figure 6.6 Energy consumption results of CDSM. 133
Figure 6.7 Varying the number of extra page faults of the SP-OMP benchmark. 134
Figure 6.8 Comparing CDSM to related work. 136
Figure 6.9 Communication behavior of the BRAMS application detected by CDSM. 137
Figure 6.10 Performance results of BRAMS on *Xeon*. 138
Figure 6.11 Overhead of CDSM. 138

Figure 7.1 Overview of kMAF. 148
Figure 7.2 Example of the update of kMAF's data structures. 151
Figure 7.3 Performance improvements on *Itanium*. 158
Figure 7.4 Performance improvements on *Xeon*. 158
Figure 7.5 QPI traffic reduction on *Xeon*. 159
Figure 7.6 Performance improvements on *Opteron*. 159
Figure 7.7 Energy consumption results on *Xeon*. 161
Figure 7.8 Running multiple applications on the *Xeon* machine. 161
Figure 7.9 Communication pattern of both versions of Ondes3D. 163
Figure 7.10 Communication pattern of HashSieve. 164
Figure 7.11 Varying the number of additional page faults of kMAF. 166
Figure 7.12 Varying the *factor* of the locality formula. 167
Figure 7.13 Performance improvements of kMAF with only thread or data mapping. 168
Figure 7.14 Performance results of SP-OMP when running with small and large pages. 169
Figure 7.15 Number of cycles of each kMAF operation on the *Xeon* machine. 170

LIST OF TABLES

Table 2.1 Overview of the parallel benchmark suites. 38
Table 2.2 Overview of the scientific applications. 39
Table 2.3 Overview of the systems used in the evaluation. 41

Table 3.1 Overview of the communication metrics introduced in this section. 61
Table 3.2 Overview of benchmark suitability for task mapping. 75

Table 4.1 Page usage of the matrix multiplication. 81
Table 4.2 Overview of the page usage metrics introduced in this section. 87
Table 4.3 Input for the data mapping example. 90
Table 4.4 Average values of the balance metrics of the mappings policies 94
Table 4.5 Dynamic page usage $Page_{dyn}$ 99
Table 4.6 Overview of benchmark suitability for data mapping. 100

Table 5.1 Configuration of the machine that was simulated in SiNUCA 105

Table 6.1 Summary of the main related task mapping mechanisms. 122
Table 6.2 Experimental configuration of CDSM. 128
Table 6.3 Overhead of the FT-MPI benchmark. 139

Table 7.1 Overview of the main related work in data mapping. 146
Table 7.2 Behavior of kMAF's data mapping policy. 154
Table 7.3 Default configuration of kMAF used in the experiments. 155
Table 7.4 Page usage statistics of Ondes3D for 4 NUMA nodes. 163
Table 7.5 Execution time of two versions of Ondes3D 164
Table 7.6 Page usage statistics of HashSieve for 4 NUMA nodes. 165
Table 7.7 Execution time of two versions of HashSieve. 165
Table 7.8 Overhead of the SP-OMP benchmark, measured on the *Xeon* machine. 170
Table 7.9 Overhead of kMAF of the benchmarks. 170

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
ASLR	Address Space Layout Randomization
BMC	Baseboard Management Controller
BRAMS	Brazilian developments on the Regional Atmospheric Modeling System
BV	Block Vector
ccNUMA	cache-coherent NUMA
CDSM	Communication Detection in Shared Memory
CFD	Computational Fluid Dynamics
CFS	Completely Fair Scheduler
CMP	Chip Multi-Processing
DBI	Dynamic Binary Instrumentation
DRAM	Dynamic Random Access Memory
DSM	Distributed Shared Memory
FLOP	Floating Point Operation
FSB	Front-Side Bus
HPC	High-Performance Computing
HPCC	High-Performance Computing Challenge
IBS	Instruction-Based Sampling
ILP	Instruction-Level Parallelism
IPMI	Intelligent Platform Management Interface
kMAF	kernel Memory Affinity Framework
LLC	Last Level Cache
LRU	Least Recently Used
LWP	Lightweight Profiling
McPAT	Multicore Power, Area, and Timing
MPE	MPI Parallel Environment
MPI	Message Passing Interface
MPSS	Multiple Page Size Support
MRU	Most Recently Used
MSE	Mean Squared Error

NAS	NASA Advanced Supercomputing
NPB	NAS Parallel Benchmarks
NUMA	Non-Uniform Memory Access
NV	NUMA Vector
OMP	OpenMP
OoO	Out-of-Order
OS	Operating System
PAE	Physical Address Extension
PARSEC	Princeton Application Repository for Shared-Memory Computers
PCM	Performance Counter Monitor
PEBS	Precise Event Based Sampling
PMU	Performance Monitoring Unit
PU	Processing Unit
QPI	QuickPath Interconnect
RAMS	Regional Atmospheric Modeling System
RAPL	Running Average Power Limits
ROB	Re-Order Buffer
ROI	Region Of Interest
SMP	Symmetric Multi-Processing
SMT	Simultaneous Multi-Threading
SPCD	Shared Pages Communication Detection
TIG	Task Interaction Graph
TLB	Translation Lookaside Buffer
TLP	Thread-Level Parallelism
TLS	Thread-Local Storage
TSC	Time Stamp Counter
UMA	Uniform Memory Access

CONTENTS

1 INTRODUCTION **21**
1.1 Parallel Shared Memory Architectures **24**
1.1.1 State-of-the-Art Architectures 24
1.1.2 Emerging Technologies 25
1.2 Measuring the Impact of Mapping **26**
1.2.1 Task and Data Mapping Example 27
1.2.2 Benefits of Improved Mappings 28
1.3 Mapping Approaches **30**
1.3.1 Classification of Memory Access Behavior 30
1.3.2 Classification of Mapping Mechanisms 31
1.4 Contributions of this Thesis **33**
1.5 Document Organization **33**

2 CHARACTERIZING PARALLEL APPLICATIONS: OVERVIEW **37**
2.1 Benchmark Selection **37**
2.1.1 Parallel Benchmark Suites 37
2.1.2 Real Scientific Applications 39
2.2 Hardware Architectures **40**
2.3 Characterization Methodology **40**
2.4 Related Work on Characterization of Memory Access Behavior **42**

3 CHARACTERIZING COMMUNICATION BEHAVIOR **45**
3.1 Qualitative Communication Behavior **45**
3.1.1 Explicit and Implicit Communication 45
3.1.2 True/False Communication and Communication Events 46
3.1.3 Read and Write Memory Accesses 48
3.1.4 Communication Direction and Communication Matrix 48
3.1.5 Relaxing the Definition of Communication 50
3.1.6 Common Types of Communication Patterns 53
3.1.7 Communication Balance 56
3.2 Quantitative Communication Behavior **57**
3.2.1 Communication Heterogeneity 58
3.2.2 Communication Balance 58
3.2.3 Communication Amount 60
3.2.4 Communication Ratio 60
3.2.5 Temporal Communication Behavior and Phase Detection 60
3.2.6 Summary of Communication Metrics 61
3.3 Task Mapping Policies **61**
3.3.1 The Policies 61
3.3.2 Example of Policy Behavior 64
3.4 Communication Behavior of the Benchmarks **65**
3.4.1 Global Communication Behavior 66
3.4.2 Dynamic Behavior 71
3.5 Summary of Communication Behavior **75**

4 CHARACTERIZING PAGE USAGE **77**
4.1 Qualitative Page Usage **77**
4.1.1 Introduction to Page Usage 77

4.1.2 Example of Page Usage 78
4.1.3 Common Types of Page Usage 79
4.1.4 Data Mapping for MPI Applications? 83
4.2 Quantitative Page Usage 83
4.2.1 Memory Access Exclusivity 83
4.2.2 Memory Access Balance 84
4.2.3 Total Memory Usage 85
4.2.4 Dynamic Behavior 85
4.2.5 Locality of a Page Mapping 86
4.2.6 Summary of Page Usage Metrics 86
4.3 Data Mapping Policies 87
4.3.1 The Policies 87
4.3.2 Example of Policy Behavior 90
4.4 Page Usage of the Benchmarks 91
4.4.1 Global Behavior 91
4.4.2 Dynamic Behavior 97
4.5 Summary of Page Usage 100

5 TASK AND DATA MAPPING POLICIES 103
5.1 Evaluating Task and Data Mapping in a Simulator 103
5.1.1 Simulation Methodology 104
5.1.2 Simulation Results 105
5.1.3 Summary 106
5.2 Evaluating Task Mapping Policies 107
5.2.1 Experimental Methodology 107
5.2.2 Results 108
5.2.3 Comparison to Predicted Task Mapping Suitability 109
5.2.4 Summary 110
5.3 Evaluating Data Mapping Policies 110
5.3.1 Methodology 110
5.3.2 Performance Results 111
5.3.3 Comparison to Predicted Data Mapping Suitability 113
5.3.4 Summary and Discussion 114
5.4 An Oracle Mapping Mechanism 114
5.5 Summary 115

6 CDSM: COMMUNICATION-AWARE TASK MAPPING 119
6.1 Introduction 119
6.2 Related Work on Task and Process Mapping 120
6.2.1 Applications Based on Message Passing 120
6.2.2 Multithreaded Applications 120
6.2.3 Summary of Related Work on Task Mapping 122
6.3 CDSM: Communication Detection in Shared Memory 122
6.3.1 Fundamentals of CDSM 123
6.3.2 Implementation 125
6.3.3 Calculate the Task Mapping 126
6.3.4 Runtime Overhead of CDSM 127
6.3.5 Summary 127
6.4 Methodology of the Experiments 127
6.5 Evaluation of CDSM 128
6.5.1 Accuracy of Detected Communication Behavior 128

6.5.2 Performance Results 130
6.5.3 Energy Consumption 133
6.5.4 Varying the Number of Page Faults 134
6.5.5 Comparing CDSM to Related Work 135
6.5.6 Case Study: BRAMS 136
6.5.7 Overhead of CDSM 137
6.6 Summary 139

7 KMAF: INTEGRATED TASK AND DATA MAPPING 141
7.1 Introduction 141
7.2 Related Work on Data Mapping 142
7.2.1 OS-Based Mapping 142
7.2.2 Compiler-Based and Runtime-Based Mapping 144
7.2.3 Hardware-Based Mapping 145
7.2.4 Manual Mapping 145
7.2.5 Summary of Related Work 146
7.3 kMAF: The Kernel Memory Affinity Framework 147
7.3.1 Overview of KMAF 147
7.3.2 Detecting Memory Access Behavior 147
7.3.3 Storage and Analysis of the Detected Behavior 149
7.3.4 Thread Mapping 151
7.3.5 Data Mapping 151
7.3.6 Supporting Multiple Running Applications 154
7.3.7 Implementation of KMAF 155
7.3.8 Overhead of KMAF 155
7.4 Methodology of the Experiments 156
7.5 Results 157
7.5.1 Single Applications 157
7.5.2 Energy Consumption 160
7.5.3 Multiple Applications 160
7.5.4 Case Study: Ondes3D 162
7.5.5 Case Study: HashSieve 164
7.5.6 Mechanism Sensitivity 165
7.5.7 Performing Thread and Data Mapping Separately 167
7.5.8 Performance Improvements with Larger Pages 168
7.5.9 Overhead of KMAF 169
7.6 Summary 170

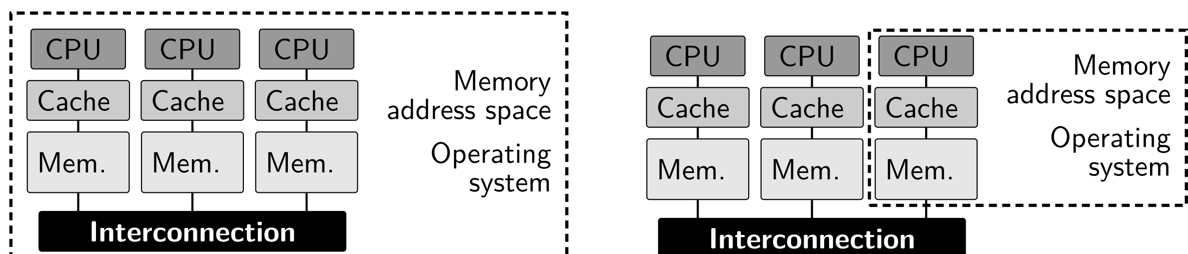
8 CONCLUDING REMARKS 173
8.1 Summary of Main Results 173
8.2 Released Software 174
8.3 Research Perspectives 174
8.4 Publications 176

REFERENCES 179

1 INTRODUCTION

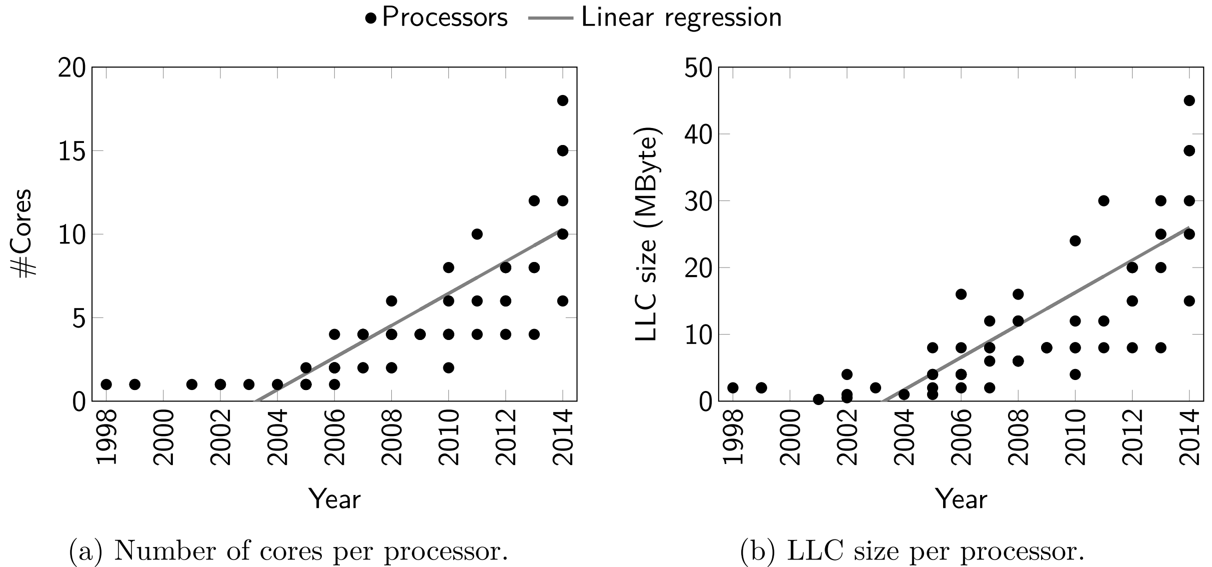
Since reaching the limits of Instruction Level Parallelism (ILP), Thread Level Parallelism (TLP) has become important to continue increasing the performance of computer systems. Increases in the TLP are accompanied by more complex memory hierarchies, consisting of several private and shared cache levels, as well as multiple memory controllers that introduce Non-Uniform Memory Access (NUMA) characteristics, to provide data to memory-hungry applications. As a result, the performance of memory accesses started to depend also on the location of the data (WANG et al., 2012; COTEUS et al., 2011). Accesses to data that is located on local caches and NUMA nodes have a higher bandwidth and lower latency than accesses to remote caches or nodes (FELIU et al., 2012). Improving the *locality* of memory accesses is mostly important when the time to access remote caches and NUMA nodes is much higher than an access to local ones (RIBEIRO et al., 2010; LIU; MELLOR-CRUMMEY, 2014). However, when the difference between local and remote accesses is low, the influence of locality becomes lower. In these cases, another important factor to be considered is the load of the caches and memory controllers. The goal is to *balance* the memory accesses such that all caches and controllers handle a similar number of requests (WANG; MORRIS, 1985; BELLOSA; STECKERMEIER, 1996; AWASTHI et al., 2010; BLAGODUROV et al., 2010; DASHTI et al., 2013). Improving locality and balance are therefore important ways to optimize performance and energy consumption of parallel applications in modern architectures.

In this thesis, we focus on *shared memory architectures*, that is, parallel architectures that share a single global memory address space and execute a single instance of an operating system. Current shared memory architectures consist of several processors, cores, and NUMA nodes, as shown in Figure 1.1a. Interconnections in this type of architecture consist of traditional buses, or newer point-to-point systems, such as Intel's QuickPath Interconnect (QPI) (ZIAKAS et al., 2010) and AMD's HyperTransport (CONWAY, 2007). A special type of shared memory system is a *distributed shared memory (DSM) architecture* (NITZBERG; LO, 1991). In these systems, the main memories are physically separate but can be addressed as a single address space. Usually, each DSM platform has a custom interconnect, such as SGI's NUMAlink interconnect (WOODACRE et al., 2005), which is used in the Altix platform. In contrast to shared memory systems, cluster systems are *distributed memory architectures*, where each cluster node has its own memory address space, as shown in Figure 1.1b. Each cluster node runs its own copy of an operating



(a) Shared memory system with 3 NUMA nodes. (b) Cluster system with 3 cluster nodes.

Figure 1.1: Comparison of a shared memory architecture and a cluster system.



(a) Number of cores per processor.

(b) LLC size per processor.

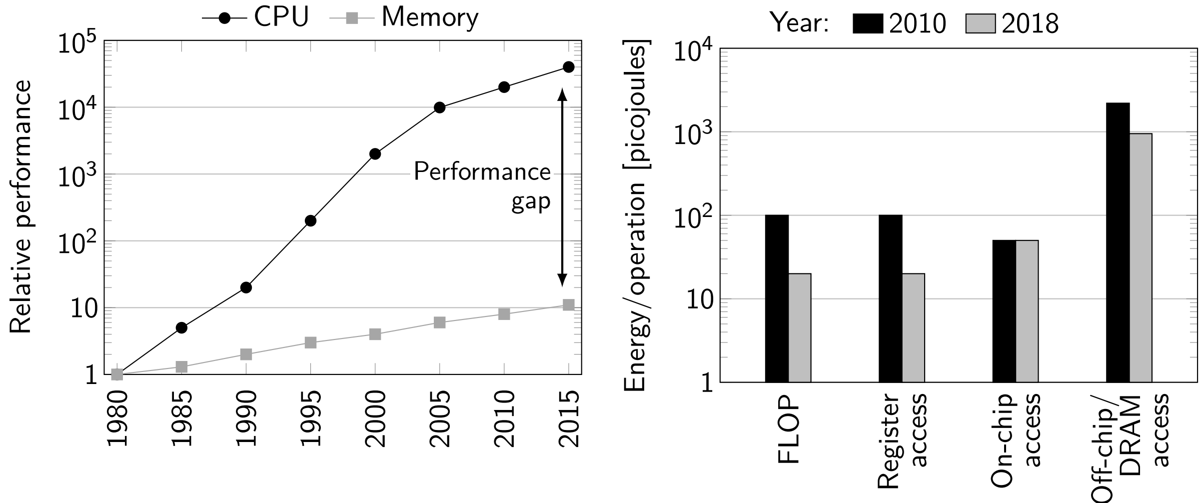
Figure 1.2: Scaling of the Intel Xeon processor between 1998 and 2014..

system and needs to communicate with other nodes through interconnections such as Ethernet or Infiniband. Although the mechanisms that will be presented in this thesis work in shared memory systems, they can be applied to cluster systems as well to optimize memory accesses in each cluster node.

Historically, parallel shared memory architectures were introduced as a solution to the high power consumption and wire-delay problems (AGARWAL et al., 2000) of single-threaded cores. TLP was increased through multiple processors per systems (Symmetric Multi-Processing, SMP), multiple cores per processors (Multi-core, or Chip Multi-Processing, CMP) and execution of multiple tasks[1] on each core simultaneously (such as Symmetric Multi-Threading, SMT). We refer to an element of a shared memory system that can execute a task as a *processing unit (PU)*. We assume that all PUs are homogeneous, that is, they have the same computational power in terms of functional units, cache sizes and execution frequency, among others. As an example of the TLP increase, Figure 1.2 shows the number of cores and Last Level Cache (LLC) size of the Intel Xeon processors released between 1998 (first Xeon generation) and 2014. Every 4 to 5 years, the number of cores and LLC size has been doubling.

This increase of the TLP leads to a higher memory pressure and exacerbates the memory wall problem (WULF; MCKEE, 1995). For these reasons, traditional bus interconnections between processors and memories (such as the Front-Side Bus, FSB) were replaced by dedicated point-to-point interconnections, resulting in a NUMA behavior. Despite these improvements, memory accesses still represent a challenge for the performance and energy consumption of applications. For the performance, the gap between CPU and memory performance still keeps on increasing (Figure 1.3a). Analyzing the energy consumption for computation and data accesses presents similar challenges. Figure 1.3b shows the average energy consumption (in picojoules) of a floating point operation (FLOP),

[1] In the context of this thesis, we use the word *task* to refer to the threads and processes of parallel applications. When it is necessary for the discussion, we will refer to threads and processes directly.



(a) The memory wall problem. Comparison of relative processor and memory performance (1980=1). Adapted from (WULF; MCKEE, 1995; METER, 2009).

(b) Energy consumption per operation, for computation (FLOP) and data accesses (all others). Adapted from (SHALF; DOSANJH; MORRISON, 2010).

Figure 1.3: The impact of memory accesses on performance and energy consumption. Note that the y-axis in both graphs is in logarithmic scale.

a register access, an on-chip cache access, and an off-chip cache access/DRAM access. Values from 2010, as well as predictions for 2018 are depicted in the figure. The results show that memory accesses to local caches and memory have a far lower energy consumption than remote accesses and are projected to remain so for the foreseeable future (SHALF; DOSANJH; MORRISON, 2010). Furthermore, moving data between processors, caches, and the memory has an energy consumption that is generally much higher than the computation performed on the data (DALLY, 2010).

In distributed-memory environments, locality has been improved by placing processes that communicate on cluster nodes that are close to each other (SISTARE; VANDEVAART; LOH, 1999; RODRIGUES et al., 2009). For parallel applications running on shared memory architectures, memory access locality can be improved in two ways. First, by executing tasks that access shared data close to each other in the memory hierarchy, they can benefit from shared caches and faster intra-chip interconnections (CRUZ; DIENER; NAVAUX, 2012; DIENER; CRUZ; NAVAUX, 2013). We refer to accesses to shared data as *communication* in this thesis and call the resulting mapping of tasks to processing units a communication-aware *task mapping*. Most parallel programming APIs for shared memory, such as OpenMP and Pthreads, directly use memory accesses to communicate. Even many implementations of the Message Passing Interface (MPI) (which uses explicit functions to communicate) contain optimizations to communicate via shared memory. Second, the memory pages that a task accesses should be placed on NUMA nodes close to where it is executing, to reduce the inter-node traffic as well as to increase the performance of accesses to the main memory (RIBEIRO et al., 2009). We call this technique *data mapping*. For both types of mapping, balancing the communication and memory accesses can also become important in order to prevent an overload.

The state-of-the-art research in this area focuses on either task or data mapping, performing them only separately. We make the case that mapping should be performed in an integrated way in order to achieve the maximum gains. Some mechanisms rely on communication or memory access traces (RODRIGUES et al., 2009; MARATHE; THAKKAR; MUELLER, 2010; DIENER et al., 2010; CRUZ et al., 2011), which cause a high overhead (ZHAI; SHENG; HE, 2011; BARROW-WILLIAMS; FENSCH; MOORE, 2009) and generate incorrect data if the behavior of the application changes between executions. Mechanisms that use indirect information about the memory access behavior, such as cache statistics or IPC (AZIMI et al., 2009; KLUG et al., 2008; BROQUEDIS et al., 2010a; RADOJKOVIĆ et al., 2013), can result in less accurate mappings. Other approaches use architecture dependent features or require hardware changes or changes to the applications themselves to perform an optimized mapping (MARATHE; MUELLER, 2006; CRUZ; DIENER; NAVAUX, 2012; CRUZ et al., 2014a; TIKIR; HOLLINGSWORTH, 2008; OGASAWARA, 2009). These approaches can not be considered generic solutions for the mapping problem.

The goal of this thesis is to improve on the current state-of-the-art by introducing mechanisms that perform an integrated task and data mapping on the kernel level. The proposed techniques use sampling of memory accesses to determine the behavior of the parallel application during its execution, and use this information to perform the mapping. The mechanisms are compatible with a wide range of hardware architectures, requiring no changes to the hardware. Since they operate on the kernel level, they require no changes to the applications, are independent of the parallelization API and support several running applications at the same time. Extensive evaluation using a large set of parallel benchmarks show significant improvements of performance and energy efficiency.

The research described in this thesis was conducted in the context of a joint degree between the Federal University of Rio Grande do Sul (UFRGS) and the Technische Universität Berlin. Parts of the research were developed at the Parallel and Distributed Processing Group (GPPD) at UFRGS and the at the Communication and Operating Systems group (KBS) at TU Berlin.

1.1 Parallel Shared Memory Architectures

Modern shared memory architectures are characterized by an increasing amount of parallelism in each system. We briefly discuss the current state-of-the-art of these architectures and give an overview of emerging technologies that affect the memory access behavior.

1.1.1 State-of-the-Art Architectures

Traditional parallel shared memory architectures were built with Symmetric Multi Processing (SMP), that is, multiple single-core processors interconnected through a shared bus. Each processor can only execute a single thread at the same time, and has a cache that is private to the CPU. Such systems contain a single memory controller, which is

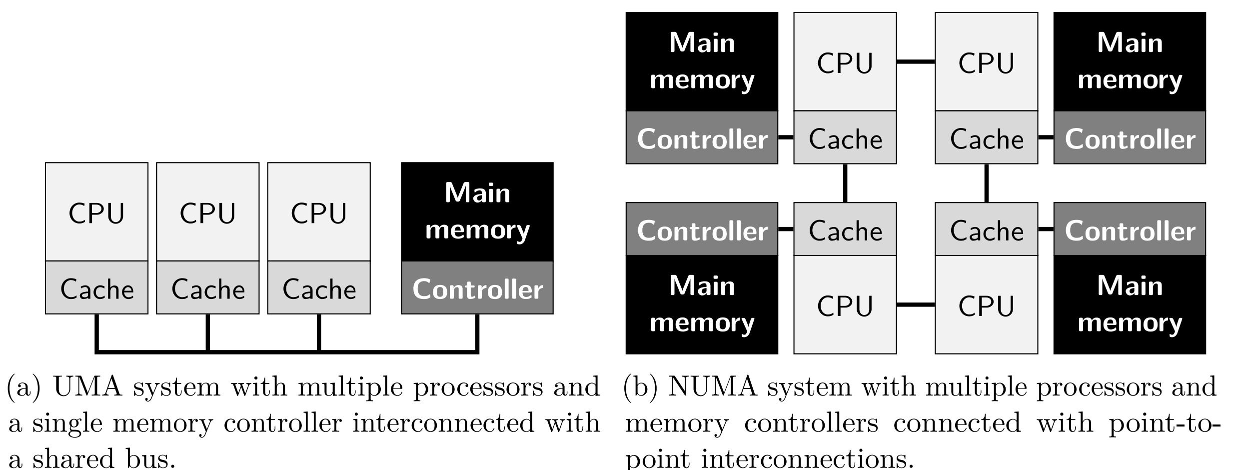
usually connected to the same bus shared between the processors. Figure 1.4a depicts an example of such a Uniform Memory Access (UMA) architecture.

Since traffic on the shared bus and accesses to the main memory represent a significant bottleneck, these UMA systems are being replaced by architectures that feature a Non-Uniform Memory Access (NUMA) behavior. In NUMA systems, the bus has been replaced by high-speed point-to-point interconnections. To increase the efficiency of memory accesses, the physical memory is split between several memory controllers, such that multiple memory requests can be handled at the same time. In this thesis, we only discuss cache-coherent NUMA architectures (ccNUMA), where the cache coherence is maintained automatically by the hardware. An example of such a system is shown in Figure 1.4b. NUMA systems are characterized by an overhead to access memory on remote memory controllers, which is called the *NUMA factor* (PILLA et al., 2011).

In addition to NUMA behavior, the parallelism in each CPU has increased remarkably compared to SMP machines. To increase the utilization of functional units, techniques such as Simultaneous Multi-Threading (SMT) allow the execution of multiple tasks concurrently by sharing the same functional units. To reduce data movement between processors, the industry has been duplicating functional units on processors, creating multi-core or CMP chips. Most current architectures employ both SMT and CMP to maximize the parallelism on each processor, resulting in the introduction of complex cache hierarchies with private and shared cache levels.

1.1.2 Emerging Technologies

Since NUMA architectures have reached the mainstream, several important developments have happened that affect the memory access efficiency and mapping policies. Most of the challenges introduced by these emerging technologies will be discussed in more detail throughout the thesis. First of all, manycore architectures will lead to a large increase of parallelism on a single chip, going from tens of cores in multi-core systems to



(a) UMA system with multiple processors and a single memory controller interconnected with a shared bus.

(b) NUMA system with multiple processors and memory controllers connected with point-to-point interconnections.

Figure 1.4: Comparison of UMA and NUMA architectures.

thousands of cores or more (ASANOVIC et al., 2006). This impacts especially the task mapping (SINGH et al., 2013), requiring the introduction of more efficient algorithms or limiting the mapping to only a part of the chip. Chapter 3 discusses some of these questions in more detail. On the other hand, the gains from task mapping are also expected to be higher due to the larger and more complex chip structure. Another technology that affects task mapping is the introduction of additional cache levels, such as L4 caches consisting of embedded Dynamic Random Access Memory (eDRAM) (WU et al., 2009) on the chip. This can impact the improvements from task mapping by further increasing the benefits of keeping data local.

For data mapping, two main trends are emerging. Due to the rising memory consumption of parallel applications, increasing the page size of modern systems is an attractive way to reduce the overhead of memory management. Larger pages reduce the number of page faults (GAUD et al., 2014), TLB misses (BASU et al., 2013), and TLB shootdowns (VILLAVIEJA et al., 2011), and require less memory to store the page table (CASCAVAL et al., 2005). For these reasons, many applications can benefit from significant performance improvements when using larger pages (WEISBERG; WISEMAN, 2009; BASU et al., 2013). Most current architectures have a default page size of a few KByte (such as 4 KByte in Intel x86 (INTEL, 2013b)) and optionally support much larger pages (256 MByte in Intel Itanium and 1 GByte in Intel x86_64, for example). Also, most modern operating systems include at least rudimentary support for larger pages, such as the Huge page mechanism for Linux (LU et al., 2006), Superpages in FreeBSD (NAVARRO, 2004), and the Multiple Page Size Support (MPSS) of Solaris (MCDOUGALL, 2004). Although larger pages increase memory management efficiency, it presents challenges for data mapping, as the larger granularity reduces opportunities for improvements (GAUD et al., 2014). This aspect will be discussed in more detail in Chapters 4 and 7.

The second trend that impacts data mapping is the increase of the number of memory controllers in shared memory systems. Most architectures currently in use contain a single memory controller per chip, such as most modern Intel Xeon processors. However, manufacturers are beginning to include multiple memory controllers on the same chip, leading to a NUMA behavior even within a single socket, as well as a hierarchy of memory controllers in a multi-socket system. This can increase the gains from an improved data mapping policy. Many current AMD processors feature multiple memory controllers (AMD, 2012), and we will evaluate the performance improvements of data mapping on such an architecture in Chapters 5 and 7.

1.2 Measuring the Impact of Mapping

To evaluate how mapping affects parallel applications, we experiment with a small synthetic benchmark to discuss the benefits of improved task and data mappings.

1.2.1 Task and Data Mapping Example

To illustrate the impact of mapping decisions on the performance and energy consumption of parallel applications, consider the NUMA system shown in Figure 1.5, which is based on the Intel Xeon Nehalem microarchitecture (INTEL, 2010b). This system consists of 4 multi-core processors with support for Simultaneous Multi-Threading (SMT). Each processor contains an L3 cache that is shared by all cores, as well as L1 and L2 caches that are private to each core. Since the processors have their own memory controllers, each one forms a NUMA node and is connected to its local main memory.

In this system, there are 3 different ways to execute a pair of tasks, labeled ⓐ, ⓑ and ⓒ in the figure, which impact the access to shared data between the tasks. In case ⓐ, the tasks execute on the same core, and can access shared data through the fast private caches as well as the local NUMA node. In case ⓑ, tasks execute on different cores in the same processor and can still benefit from the shared L3 cache and fast on-chip interconnection. In case ⓒ however, tasks need to access shared data across the slow off-chip interconnection. Furthermore, since the shared data needs to be placed on one of the NUMA nodes, the task that runs on the other node has a further penalty to access the memory located on the remote node.

To evaluate the impact of mapping choices, we developed a producer/consumer benchmark that consists of two tasks. The producer task repeatedly writes data to a shared vector, which gets read by the consumer task. Neither task performs significant calculations apart from incrementing a loop counter, and the application is therefore highly memory-bound. We executed this benchmark 20 times on the machine described above, using the default task and data mapping policies of the Linux operating system (OS), and measured the execution time and energy consumption of each run.

The results of this experiment are shown in Figure 1.6. For each of the 20 executions,

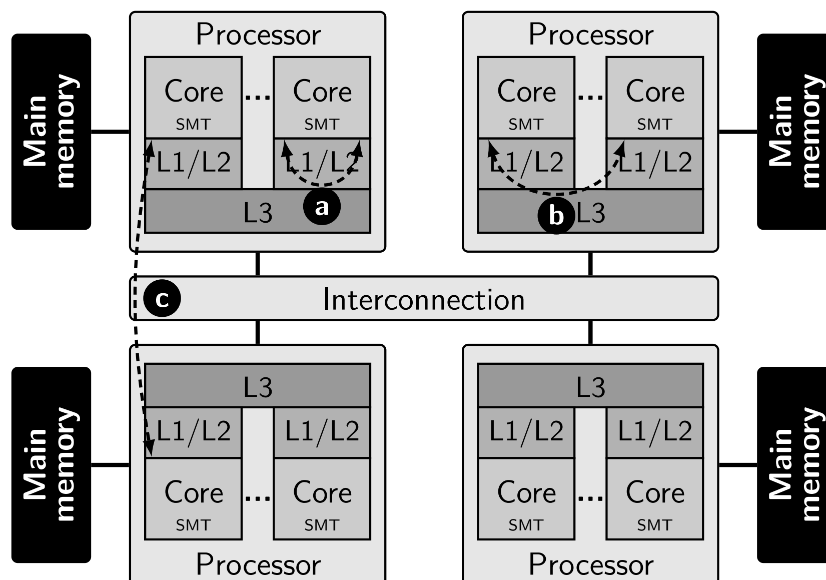


Figure 1.5: Example NUMA architecture with 4 NUMA nodes. In this system, two tasks can access shared data in three different ways, labeled ⓐ, ⓑ and ⓒ.

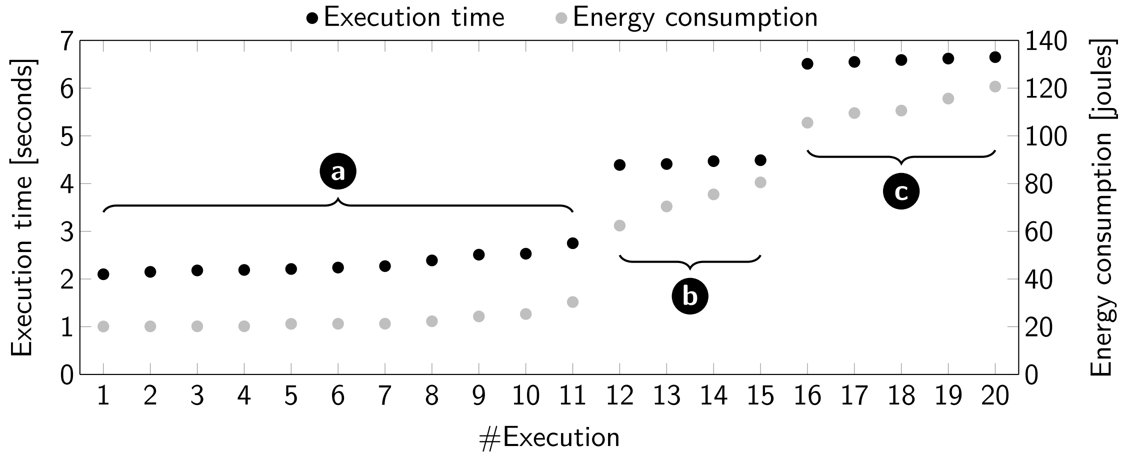


Figure 1.6: Performance and energy consumption results of 20 executions of the producer/consumer benchmark running on the machine shown in Figure 1.5. The results are ordered according to the execution time of the benchmark. ⓐ, ⓑ, and ⓒ correspond to the three mapping decisions depicted in Figure 1.5, performed by the OS.

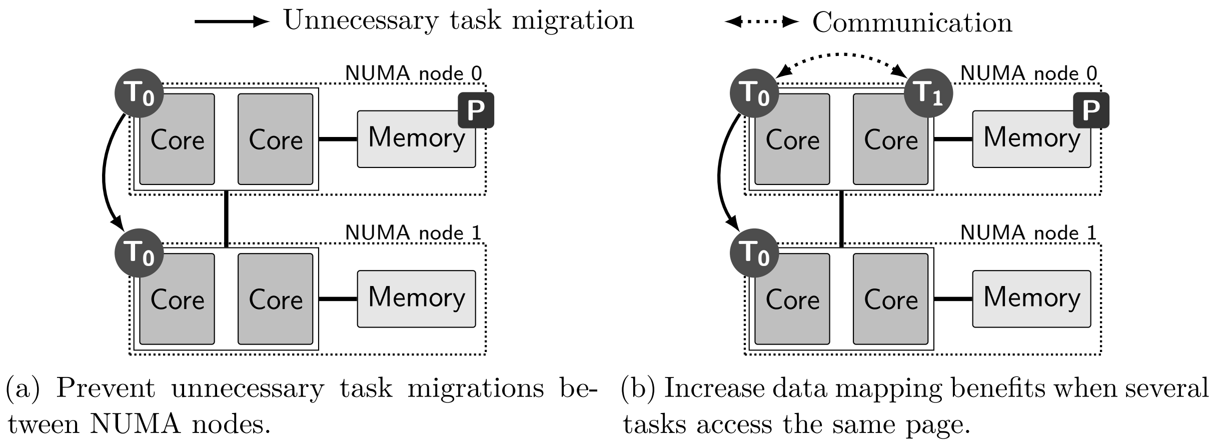
we show the execution time and energy consumption, and order the results according to the execution time. From the results, the three mapping possibilities described before, ⓐ, ⓑ, and ⓒ, can be clearly determined. For about half of the executions, the OS chose the correct mapping for this particular benchmark, executing the two tasks on the same core, resulting in the lowest execution time and energy consumption. Since the OS uses the first-touch policy to allocate pages, the data mapping is also correct in this case. For the other half of the executions, the OS placed the tasks farther apart, reducing performance and energy efficiency due to the worse task mapping. Furthermore, when the tasks get mapped to different NUMA nodes (case ⓒ), they have to perform memory accesses to the remote NUMA nodes, which contributes to the further decreases in this case. Between the best and worst mapping, execution time more than tripled, and energy consumption was increased 5.5 times. These results show that mapping has a large impact on the application's efficiency.

1.2.2 Benefits of Improved Mappings

Task and data mapping aim to improve the memory accesses to shared and private data in parallel applications. This section discusses how mapping can improve performance and energy efficiency.

1.2.2.1 Performance Improvements

Task mapping improves the efficiency of the interconnections, reducing inter-chip traffic that has a higher latency and lower bandwidth than intra-chip interconnections. It also reduces the number of cache misses of parallel applications. In read-only situations, executing tasks on the same shared cache reduces data replication, thereby increasing the cache space available to the application (CHISHTI; POWELL; VIJAYKUMAR, 2005). In



(a) Prevent unnecessary task migrations between NUMA nodes.

(b) Increase data mapping benefits when several tasks access the same page.

Figure 1.7: Two scenarios that show why task mapping is a requirement for data mapping. Consider that two tasks T_0 and T_1 access a page P.

read-write or write-write situations, an optimized task mapping also reduces cache line invalidations, reducing the traffic on the interconnections as well as preventing a cache miss on the next access to the cache line (ZHOU; CHEN; ZHENG, 2009). Data mapping improves the memory locality on NUMA machines by reducing the number of accesses to remote memory banks. As task mapping, it improves the efficiency of the interconnections by reducing the traffic between NUMA nodes. This increases the memory bandwidth available in the system and reduces the average memory access latency.

It is important to note that task mapping is a prerequisite for data mapping, for the two reasons depicted in Figure 1.7, where two tasks T_0 and T_1 access a page P. First, task mapping prevents unnecessary task migrations between NUMA nodes, such that tasks can benefit from the local data accesses (Figure 1.7a). In the figure, task T_0 is migrated unnecessarily between NUMA nodes, such that accesses to page P are now accesses to a remote NUMA node, rendering the data mapping of the page ineffective. Second, data mapping alone is not able to improve locality when more than one task accesses the same page, since the tasks may be executing on different NUMA nodes. In this situation, only the tasks that are executing on the same node where the data is located can benefit from the increased locality (Figure 1.7b). By performing the task mapping, tasks that communicate a lot are executed on the same NUMA node, thereby improving the effectiveness and the gains of the data mapping. We will evaluate the interaction of the two types of mapping in more detail throughout this thesis.

1.2.2.2 Energy Consumption Improvements

As shown in the experiment with the producer/consumer benchmark, improved task and data mappings can also reduce energy consumption of parallel applications. By reducing application execution time, static energy consumption (leakage) will be reduced proportionally in most circumstances. Reducing the number of cache misses and traffic on the interconnections reduces the dynamic energy consumption, leading to a more energy-efficient execution of parallel applications due to improved mappings. We therefore expect for most experiments a reduction of energy consumption that is similar to the

performance improvements and will evaluate our mapping mechanisms in this regard as well.

1.3 Mapping Approaches

Task and data mappings can be generated and applied in different ways. The type of mapping determines the amount of benefits, the overhead, as well as the applicability to different classes of applications, depending on the dynamicity of their memory access behavior. This section briefly compares parallel applications in terms of this dynamicity and afterwards presents a classification of mapping approaches.

1.3.1 Classification of Memory Access Behavior

On a high abstraction level, we classify the general memory access behavior of an application as *static* or *dynamic*, as shown in Figure 1.8. We further divide dynamic access behavior into two types. The first type is characterized by dynamic behavior *between executions*, where the behavior depends on specified parameters of the application, such as the input data or the number of tasks that will be created. Moreover, memory addresses can also change between executions, due to security techniques such as Address Space Layout Randomization (ASLR) (MARCO-GISBERT; RIPOLL, 2014) or a different order of dynamic memory allocations with functions such as `malloc()`.

The second type of dynamic behavior occurs *during the execution* of the application, due to the way a parallel algorithm is implemented (such as using work-stealing (BLUMOFE; LEISERSON, 1994) or pipeline programming models), or due to the creation and destruction of tasks or allocation/deallocation of memory. If none of these cases occur, we classify the applications' behavior as static. It is important to mention the influence of the granularity on the classification. If looking at a very small granularity, such as classifying every memory access, most applications will have a very dynamic behavior. Since we are mostly interested in the effects of mapping, we will use larger time frames for classifying the behavior.

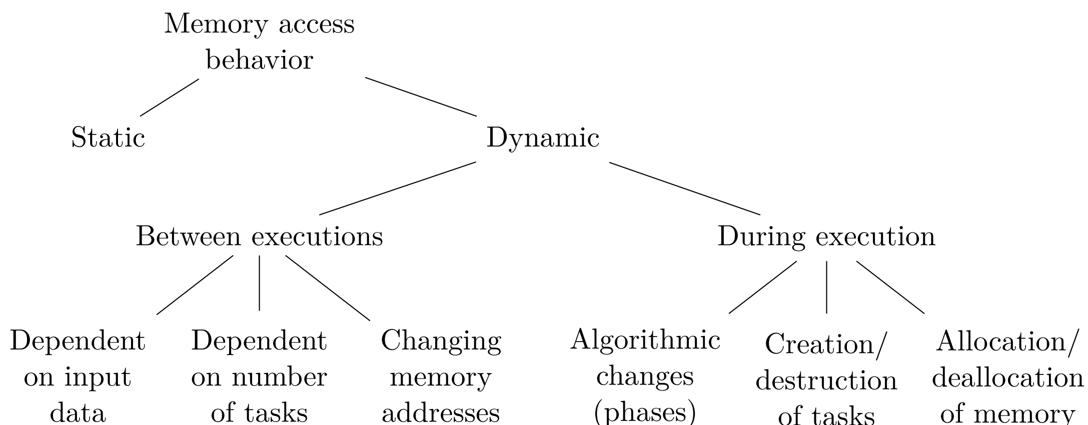


Figure 1.8: Classification of the memory access behavior of parallel applications.

The memory access classification is important for the types of mapping that can be performed. If the memory access behavior is static, no runtime migrations of tasks and memory pages need to be performed. Moreover, the behavior can be classified and analyzed through communication or memory access traces, and only the global behavior throughout the complete execution needs to be taken into account. For applications that show a dynamic behavior only during the execution, traces can still be used to analyze their behavior. However, the changing behavior during execution needs to be taken into account when performing mapping, which can require task or data migrations during the execution to achieve optimal gains. For applications whose behavior changes between executions, trace-based mechanisms require the generation of a new trace for each set of input parameters, as the detected behavior would otherwise not be valid for future executions of the application. Care must also be taken to limit the impact of changes to memory addresses. Online mechanisms directly support all types of dynamic behavior.

1.3.2 Classification of Mapping Mechanisms

Mapping mechanisms generally consist of two parts: they need to analyze and describe memory access behavior, which we call the *analysis* part. Important characteristics of the analysis are when the analysis is performed (whether information is available before execution starts), which metrics are used to describe behavior, on which level information is gathered (hardware, OS, application, ...), and if the hardware or software need to be modified. Based on the analyzed behavior, a mapping mechanism needs to apply a *policy* to determine where tasks and data should be placed and when they should be migrated. The policy can be characterized in terms of its goals (such as improving locality or balance), when it is applied (before or during execution), and if the application or runtime environment needs to be modified. Based on how the analysis and policy parts are performed in the context of the execution of a parallel application and their characteristics, we develop a classification of mapping mechanisms that consists of three groups: *manual*, *semi-automatic* and *automatic* mechanisms.

1.3.2.1 Manual Mechanisms

Manual mapping mechanisms are defined as mechanisms where the mapping is performed by the developer through source code modifications or by the user through options in the runtime environment. Libraries such as `hwloc` (BROQUEDIS et al., 2010b) and `libnuma` (KLEEN, 2004) provide information about the hardware and memory hierarchy and can be used directly from the application. The `numactl` program (KLEEN, 2004) provides options for the user to modify the data mapping of an application, such as a forced allocation on a specific NUMA node or an interleaving policy.

To perform task mapping, application developers can use OS functions to execute tasks on a specified set of processing units. For application users, many runtime environments for OpenMP and MPI offer options to specify a task mapping. Many environments also provide more high-level task mapping options, such as a *compact* mapping, where neighboring tasks are mapped such that they execute on nearby processing units (INTEL,

2012b; ARGONNE NATIONAL LABORATORY, 2014b). The most common way to perform manual data mapping is by performing an explicit memory access at the beginning of execution such that each memory page is accessed first by the task that will access it, which can be beneficial in *first-touch* data mapping policies. In case the memory access behavior changes during execution, most operating systems include functions to migrate memory pages during execution. More sophisticated libraries such as MAi (RIBEIRO et al., 2009) and libnuma (KLEEN, 2004) provide support for data mapping in the application.

Manual mapping mechanisms put the burden of calculating and performing the mapping on the developer, and can therefore not be considered a general solution to the mapping problem. They are intrusive, as they require changes to the source code of every application and adaptation to different hardware architectures. Furthermore, manual mapping can present problems if multiple applications are executing at the same time, as the applications are unaware of each other and their mapping decisions might interfere. However, they can provide the highest improvements in case the developer or the user have perfect knowledge of the application's behavior.

1.3.2.2 Semi-Automatic Mechanisms

Semi-automatic mapping mechanisms consist of two steps: First, the application is profiled to determine its memory access behavior, for example through memory or communication traces. Tools such as eztrace (TRAHAY et al., 2011) and the Pin dynamic binary instrumentation tool (LUK et al., 2005) can be used for this purpose. The profile is analyzed to determine an optimized mapping. In the second step, the calculated mappings are then applied during the real execution of the application. The profiling phase is potentially time-consuming and is not applicable if the application changes its behavior between executions. Similar to the manual mapping, it can also present problems when multiple applications are executing at the same time. Furthermore, the data generated during profiling might be very large, necessitating a time-consuming analysis (ZHAI; SHENG; HE, 2011). However, it requires no changes to the applications and incurs only a minimal runtime overhead, as the behavior analysis and mapping decisions are performed before the application starts.

1.3.2.3 Automatic Mechanisms

Automatic mapping mechanisms perform the mapping online, during the execution of the parallel application, using information gathered only during execution. This has several consequences. The advantages are that automatic mapping presents a truly general solution to the mapping problem, as concurrently executing applications can be taken into account, as well as changing behavior between or during executions. Moreover, no expensive analysis before execution has to be performed. On the other hand, they have two important issues that need to be resolved. The main challenge for this type of mechanism is the runtime overhead, as the information gathering and migration may have a large impact on the application, reducing or even eliminating the improvements of mapping. Furthermore, since no prior information about the application behavior is available, future behavior needs to be predicted using past behavior. This also implies that improvements

can be lower in comparison to mechanisms that already have this information before the application starts. We will focus on automatic mapping mechanisms in this thesis, and compare their improvements to semi-automatic and manual mechanisms.

1.4 Contributions of this Thesis

The main objective of this thesis is to improve the performance and energy consumption of modern shared memory architectures by performing automatic task and data mapping. The primary contributions of this thesis are as follows:

Characterization of Parallel Applications. We develop metrics and a methodology to describe the memory access behavior of parallel applications, focusing on the communication and page usage of the tasks during the execution. The behavior is analyzed to determine their suitability for task and data mapping.

Trace-Based Benchmark Analysis and Mapping Improvements. We analyze several parallel benchmark suites that use different parallelization paradigms and evaluate their improvements using task and data mapping using a trace-based mechanism, which will serve as the baseline for our automatic mapping mechanisms.

Automatic Kernel-Based Mechanisms to Perform Mapping. We introduce two automatic, kernel-based mechanisms to perform task and data mapping, *CDSM* and *kMAF*, and implement them in the Linux kernel. CDSM performs task mapping of parallel applications, while kMAF performs an integrated task and data mapping. Both mechanisms gather information about the memory access behavior of applications during their execution through tracing and analyzing the applications' page faults, which causes only a small overhead.

1.5 Document Organization

The remainder of this thesis is separated into two parts. The **first part** discusses which types of memory access behavior can be improved with different mapping policies and proposes a trace-based mechanisms to generate these policies. This first part consists of the following chapters. **Chapter 2** briefly presents the general methodology of the thesis, including the benchmark suites and parallel machines used for measurements, as well as the software tools used for the characterization. **Chapter 3** discusses qualitative and quantitative aspects of communication in parallel applications, introduces metrics and methodologies to describe communication behavior and presents task mapping policies that can improve different behaviors. The chapter finishes with the communication characterization of the parallel applications. **Chapter 4** introduces a similarly-structured analysis of the page usage of parallel applications and discusses various data mapping policies for improving page usage on NUMA architectures. **Chapter 5** evaluates the performance and energy consumption impact of the various task and data mapping policies and presents an Oracle mapping mechanism based on memory access traces.

The **second part** contains the main contributions of this thesis, presenting two automatic mapping mechanisms, ***CDSM*** and ***kMAF***, which are based on the concepts

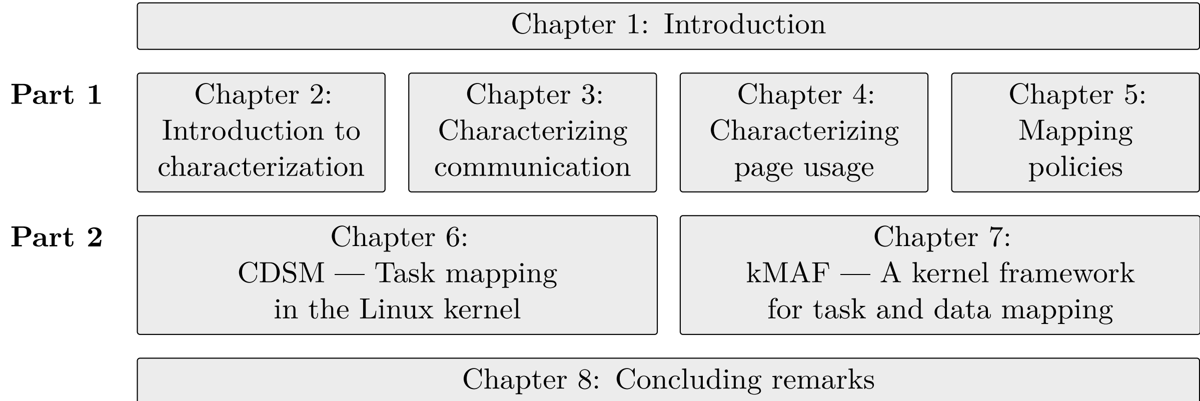


Figure 1.9: Overview of the chapters of the thesis.

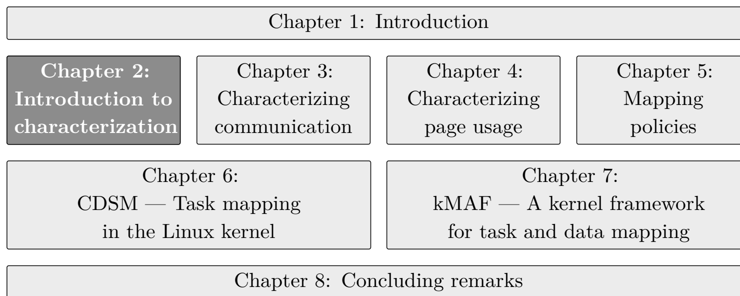
introduced in the first part. It consists of the following chapters. **Chapter 6** introduces *Communication Detection in Shared Memory (CDSM)*, an automatic task mapping mechanism for the Linux kernel. After a detailed description of CDSM, we evaluate it in terms of accuracy, as well as performance and energy consumption improvements, comparing CDSM to the Oracle and other mechanisms. **Chapter 7** introduces the *kernel Memory Affinity Framework (kMAF)*. kMAF is a generalization of CDSM, supporting data mapping in addition to task mapping and allowing the use of different types of policies.

Related work for each topic will be discussed in their respective chapters, in Section 2.4 regarding characterization of memory access behavior, Section 6.2 regarding task mapping and Section 7.2 regarding data mapping. Finally, **Chapter 8** summarizes our conclusions, gives an overview of the software produced as part of this research and outlines ideas for future work that can be based on this thesis. An overview of the structure of the thesis is shown in Figure 1.9.

Part 1:
The Memory Access Behavior of Parallel Applications

	Chapter 1: Introduction			
Part 1	Chapter 2: Introduction to characterization	Chapter 3: Characterizing communication	Chapter 4: Characterizing page usage	Chapter 5: Mapping policies
Part 2	Chapter 6: CDSM — Task mapping in the Linux kernel		Chapter 7: kMAF — A kernel framework for task and data mapping	
	Chapter 8: Concluding remarks			

2 CHARACTERIZING PARALLEL APPLICATIONS: OVERVIEW



In order to determine if applications can benefit from task or data mapping, two questions about their memory access behavior need to be addressed. Which type of behavior can be improved with mapping? How can the behavior be described mathematically? This first part of this thesis aims to answer these questions for a large set of parallel benchmarks to determine their suitability for mapping improvements, focusing on the way that they communicate and how they access memory pages. This chapter contains a brief overview about related work regarding the memory access characterization of parallel applications, introduces the benchmarks that we evaluate, and presents an overview of the memory tracer that is used to perform the characterization. In Chapter 3, we introduce metrics for the communication behavior and evaluate the benchmarks with these metrics. Similarly, Chapter 4 presents the metrics and evaluation of the page usage.

2.1 Benchmark Selection

In this thesis, we evaluate a diverse set of parallel applications that use several parallelization models and have different memory access behaviors. Experiments were performed with multiple parallel benchmark suites, as well as larger scientific applications. This section gives an overview of these applications.

2.1.1 Parallel Benchmark Suites

We selected the following parallel benchmark suites: NAS-OMP, PARSEC, NAS-MPI, HPCC and NAS-MZ. NAS-OMP and PARSEC represent typical parallel shared memory applications and are therefore the main focus in this thesis. The other benchmark suites are implemented with MPI. We use the MPICH2 framework (GROPP, 2002) MPI framework for compilation and execution of the MPI benchmarks. All benchmarks were compiled with gcc, version 4.6, with the default compilation options specified by each suite (which, in most cases, specify the `-O2` optimization level). Table 2.1 contains a summary of the parallel APIs, benchmark names, input sizes and average memory usage of the benchmarks in each suite. Memory usage is presented for benchmark configurations with 64 tasks and only includes memory that was actually accessed by the application (on the page size granularity).

NAS-OMP (JIN; FRUMKIN; YAN, 1999) is the OpenMP version of the NAS Parallel Benchmarks (NPB), which consists of 10 applications from the High Performance Computing (HPC) domain. Due to its good scaling behavior, we will execute these applications with various input sizes, *W*, *A*, *B*, *C*, and *D*, from smallest to largest, to

Table 2.1: Overview of the parallel benchmark suites used in the evaluation, showing parallelization API, benchmark names, input size and average memory usage per application.

Benchmark suite	Parallel API	Benchmark names	Input size	Memory usage
NAS-OMP v3.3.1	OpenMP	BT-OMP, CG-OMP, DC-OMP, EP-OMP, FT-OMP, IS-OMP, LU-OMP, MG-OMP, SP-OMP, UA-OMP (10)	W–D (DC-OMP: W–B)	70.1 MByte – 24.8 GByte
PARSEC v3.0 beta	Pthreads, OpenMP	Blackscholes, Bodytrack, Facesim, Ferret, Freqmine, Raytrace, Swaptions, Fluidanimate, Vips, X264, Canneal, Dedup, Streamcluster (13)	*native*	3.2 GByte
NAS-MPI v3.3.1	MPI	BT-MPI, CG-MPI, DT-MPI, EP-MPI, FT-MPI, IS-MPI, LU-MPI, MG-MPI, SP-MPI (9)	B	1.7 GByte
HPCC v1.4.3	MPI	HPCC (single application consisting of 16 workloads)	4000×4000 matrix	19.3 GByte
NAS-MZ v3.3.1	MPI + OpenMP	BT-MZ, LU-MZ, SP-MZ (3)	B	230 MByte

evaluate how the behavior changes with different inputs. The highest input size of DC-OMP is B.

PARSEC (BIENIA et al., 2008) is a suite of 13 benchmarks that focus on emerging parallel workloads for modern multi-core architectures. PARSEC benchmarks are implemented with OpenMP and Pthreads. Most applications have a highly dynamic behavior. This dynamic behavior includes creating and stopping threads during execution, as well as allocating and deallocating memory dynamically All benchmarks were executed with the *native* input set, which is the largest input set available. We measure the full execution time of each PARSEC benchmark, not just the time spent in the parallel phase (referred to as Region Of Interest (ROI) in the PARSEC documentation and source code). This is done to use an evaluation scenario resembling real program behavior (BIENIA et al., 2008; SOUTHERN; RENAU, 2015) and to ensure better comparability of the results to the other benchmarks.

NAS-MPI (BAILEY et al., 1991; BAILEY et al., 1995) is the MPI implementation of the NAS Parallel Benchmarks (NPB), consisting of 9 HPC applications. All NAS-MPI applications are executed with the B input size. We use the BlackHole (BH) variant of the DT-MPI benchmark.

HPCC (High Performance Computing Challenge) (LUSZCZEK et al., 2005) is a single application that consists of 16 different HPC workloads, such as HPL (PETITET et al., 2012) and STREAM (MCCALPIN, 1995). It is implemented in MPI. We execute

Table 2.2: Overview of the scientific applications used in the evaluation, showing the section in which they are discussed, the parallelization API, benchmark names, input size and average memory usage per application.

Application	Section	Parallel API	Input size	Memory usage
Ondes3D	7.5.4	OpenMP	$300 \times 300 \times 300$, 1000 iterations	1.2 GByte
HashSieve	7.5.5	OpenMP	Lattice of dimension 80	34.6 GByte
BRAMS	6.5.6	MPI	`lightgr`	1.9 GByte

HPCC with a square input matrix of 4000×4000 elements, leaving the other input parameters at their default.

NAS-MZ (VAN DER WIJNGAART; JIN, 2003) is the Multi-Zone implementation of the NAS benchmarks. It uses a hybrid parallelization using the MPI and OpenMP models and can be executed with varying combinations of processes and threads. For the experimental evaluation, we will execute the NAS-MZ benchmarks with 1 process per socket and a number of threads equal to the number of PUs in each socket. All NAS-MZ applications are executed with the B input size.

2.1.2 Real Scientific Applications

In addition to the benchmarks, we also analyze several larger scientific applications, to show how our mechanisms handle real-world scenarios. An overview of these applications is shown in Table 2.2.

Ondes3D[1] simulates the propagation of seismic waves and implements the fourth-order finite-differences numerical method for solving the elastodynamics equations (DUPROS et al., 2008). A review of this numerical stencil can be found in Aochi et al. (2013).

HashSieve[2] is an algorithm for the Shortest Vector Problem (SVP) (MICCIANCIO, 2002), a key problem in lattice-based cryptography. It requires large amounts of memory and has a highly irregular memory access pattern. An overview of the implementation of HashSieve can be found in Mariano et al. (2015).

BRAMS (Brazilian developments on the Regional Atmospheric Modeling System) (FREITAS et al., 2009) is the extended version of the RAMS (Regional Atmospheric Modeling System) weather prediction model (PIELKE et al., 1992). BRAMS is parallelized with MPI. We evaluated BRAMS with the `light1gr` input set[3], which represents a test case for the correctness and performance of BRAMS.

[1]The author thanks Fabrice Dupros for providing him with Ondes3D and helping with its analysis.
[2]The author thanks Artur Mariano for providing him with HashSieve and helping with its analysis.
[3]<http://brams.cptec.inpe.br/~rbrams/light/light1gr.tar.gz>

2.2 Hardware Architectures

We evaluate the parallel applications on three real NUMA machines: *Itanium*, *Xeon*, and *Opteron*. Of the three machines, *Xeon* is our main evaluation system. Results on the other systems will only be presented if they deviate substantially from the results on *Xeon*.

The *Itanium* machine represents a traditional NUMA architecture based on the SGI Altix 450 platform (WOODACRE et al., 2005). It consists of 2 NUMA nodes, each with 2 dual-core Intel Itanium 2 processors (Montecito microarchitecture (INTEL, 2007)) and a proprietary SGI interconnection, NUMAlink (WOODACRE et al., 2005). Each core has private L1, L2 and L3 caches.

The *Xeon* machine represents a newer generation NUMA system with high-speed interconnections. It consists of 4 NUMA nodes with 1 eight-core Intel Xeon processor each (Nehalem-EX microarchitecture (INTEL, 2010b)) and a QuickPath Interconnect (QPI) interconnection (ZIAKAS et al., 2010) between the nodes. Each core has private L1 and L2 caches, while the large L3 cache is shared among all the cores in the processor.

The *Opteron* machine represents a recently introduced generation of NUMA systems with multiple on-chip memory controllers. It consists of 4 AMD Opteron processors (Abu Dhabi microarchitecture (AMD, 2012)), each with 2 memory controllers, forming 8 NUMA nodes in total. Each core has a private L1 data cache, while the L1 instruction cache and L2 cache is shared between pairs of cores. The L3 cache is shared among 8 cores in the same processor.

The *Itanium* machine can execute up to 8 tasks concurrently, while *Xeon* and *Opteron* can execute 64 tasks concurrently. In our experiments, we will run each parallel application with at least this number of tasks. Such a configuration results in the best performance for the applications we evaluated and is the most common usage scenario (for example, most OpenMP runtime environments will create as many threads as PUs by default).

A summary of the machines is shown in Table 2.3. The NUMA factors of the machines, which represent the overhead of memory accesses to remote NUMA nodes compared to the local node, were measured with Lmbench (MCVOY; STAELIN, 1996). They are calculated with Equation 2.1, where m and n represent different NUMA nodes (PILLA et al., 2011).

$$\begin{aligned} \textit{NUMA factors (m,n)} &= \frac{\textit{memory read latency from m to n}}{\textit{memory read latency on m}} \\ \textit{NUMA factor (machine)} &= \max(\textit{NUMA factors}) \end{aligned} \tag{2.1}$$

2.3 Characterization Methodology

For an accurate analysis of the memory access behavior of parallel applications, it is necessary to gather the most information about the memory accesses of each task with as little overhead as possible. Information that is gathered usually consists of the memory

Table 2.3: Overview of the systems used in the evaluation.

Name	Property	Value
Itanium	NUMA	2 nodes, 2 processors/node, NUMA factor 2.1
	Processors	4× Intel Itanium 2 9030, 1.6 GHz, 2 cores, no SMT
	Caches per proc.	16 KB+16 KB L1, 256 KB L2, 4 MB L3
	Memory	16 GB DDR-400, page size 16 KB
	Operating system	Debian 7, Linux kernel 2.6.32, 64 bit
Xeon	NUMA	4 nodes, 1 processor/node, NUMA factor 1.5
	Processors	4× Intel Xeon X7550, 2.0 GHz, 8 cores, 2-way SMT
	Caches per proc.	8× 32 KB+32 KB L1, 8× 256 KB L2, 18 MB L3
	Memory	128 GB DDR3-1066, page size 4 KB
	Operating system	Ubuntu 12.04, Linux kernel 3.8, 64 bit
Opteron	NUMA	8 nodes, 2 nodes/processor, NUMA factor 2.8
	Processors	4×AMD Opteron 6386, 2.8 GHz, 8 cores, 2-way SMT
	Caches per proc.	8× 16 KB+64 KB L1, 8× 2 MB L2, 2× 6 MB L3
	Memory	128 GB DDR3-1600, page size 4 KB
	Operating system	Gentoo, Linux kernel 3.8, 64 bit

address, task ID, operation (read or write), and number of bytes of the operation of each access. Many previous memory tracing techniques are based on simulating applications with tools such as Simics (MAGNUSSON et al., 2002) and Gem5 (BINKERT et al., 2011). However, simulation has generally a very high overhead, which limits the input sizes and degree of parallelism that can be evaluated. Other mechanisms have tried to solve the overhead issue through the use of hardware counters that can sample memory accesses performed by applications. Examples of such tools include MemProf (LACHAIZE; LEPERS; QUÉMA, 2012) and Memphis (MCCURDY; VETTER, 2010). Due to the use of sampling, these tools have a lower accuracy that might result in drawing wrong conclusions about application behavior.

Another approach is to use Dynamic Binary Instrumentation (DBI) (NETHERCOTE, 2004), where a binary application is executed with added instrumentation code that performs a certain action when a particular event, such as a memory access, occurs in the application. DBI can have an overhead that is sufficiently low for accurate memory access analysis of large applications (UH et al., 2006). Three DBI frameworks that have been widely used in the community are Valgrind (NETHERCOTE; SEWARD, 2007; NETHERCOTE, 2004), MemTrace (PAYER; KRAVINA; GROSS, 2013), and Pin (BACH et al., 2010; LUK et al., 2005). Valgrind already contains CacheGrind (VALGRIND DEVELOPERS, 2014a), a tool that can be used to analyze the memory access behavior and simulate cache memories. However, Valgrind has a large drawback for parallel applications, since it serializes all threads (VALGRIND DEVELOPERS, 2014b), resulting in a reduced accuracy as well as a longer execution time. On the other hand, MemTrace directly supports parallel applications and has a low overhead, with the authors mentioning an average overhead of 2× (PAYER; KRAVINA; GROSS, 2013). However, MemTrace only supports 32-bit applications executing on 64-bit x86 machines, since it makes use of the

extra registers available in 64-bit mode to perform the instrumentation. Our applications can only be compiled in 64-bit mode due to their large memory usage and are therefore not supported by MemTrace.

For these reasons, we developed a custom memory tracer based on Pin, *numalize*, for the application characterization as part of this thesis. Numalize is built on the Pin framework, but has no other external dependencies[4]. Numalize is supported on Linux for the Intel x86 (INTEL, 2013b) and Itanium (INTEL, 2010a) hardware architectures. Numalize records all memory accesses of all tasks, storing the address, task ID and time stamp (via the Time Stamp Counter (TSC)) of each access. For each access, an analysis routine for communication or page usage is executed. These routines will be described in Chapters 3 and 4, respectively. To reduce the overhead of numalize, no information is stored on the disk during execution, everything is kept in main memory. At the end of execution, the generated behaviors are written to disk.

2.4 Related Work on Characterization of Memory Access Behavior

Related work that characterizes communication mostly focuses on applications that use explicit message passing frameworks, such as MPI. Examples include (FARAJ; YUAN, 2002; LEE, 2009; KIM; LILJA, 1998). A characterization methodology for communication is presented in (CHODNEKAR et al., 1997; SINGH; ROTHBERG; GUPTA, 1994), where communication is described with temporal, spatial and volume components. We use similar components to describe communication, but apply them in the context of shared memory, where communication is performed implicitly through memory accesses to memory areas that are shared between different tasks. Barrow-Williams et al. (2009) perform a communication analysis of the PARSEC and Splash2 benchmark suites. They focus on communication on the logical level and therefore only count memory accesses that really represent communication, filtering out memory accesses that occur due to register pressure for example. As we are interested in the architectural effects of communication, we take into account all memory accesses for the characterization.

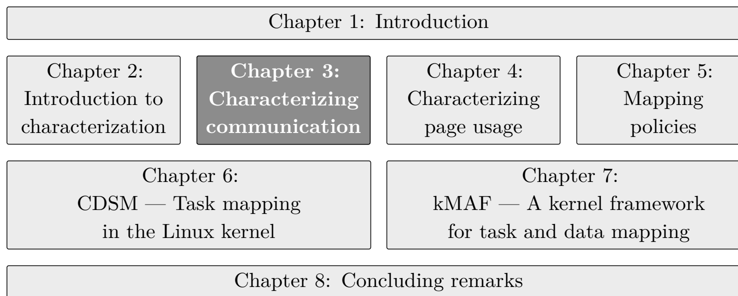
Regarding the page usage for data mapping, Majo et al. (2013) identify shared data as a challenge for improved performance on NUMA systems, as it increases the amount of memory accesses to remote NUMA nodes. They modify four applications from PARSEC to reduce the amount of data that is shared between tasks to improve the performance. In this thesis, we adopt a different approach by optimizing the mapping instead of modifying the applications. SIGMA (DEROSE et al., 2002) is a simulation infrastructure to analyze memory access bottlenecks of sequential applications, but focuses on improving cache and TLB usage without addressing NUMA issues. Most previous tools that focus on memory access analysis for NUMA characterization, such as MemProf (LACHAIZE; LEPERS; QUÉMA, 2012), MemAxes (GIMÉNEZ et al., 2014), a NUMA extension (LIU; MELLOR-CRUMMEY, 2014) for the HPCToolkit (ADHIANTO et al., 2010), and Memphis (MCCURDY; VETTER, 2010), use sampling methods such as those provided via Instruction-Based Sampling (IBS) (DRONGOWSKI, 2007) on recent

[4] Numalize is available at <https://github.com/matthiasdiener/numalize>

AMD architectures and Precise Event Based Sampling (PEBS) (LEVINTHAL, 2009) on Intel processors. Sampling reduces the accuracy of the gathered data and can therefore lead to wrong conclusions regarding the behavior. Furthermore, these tools do not treat the issue of communication. To the best of our knowledge, no systematic evaluation of the page access behavior of parallel applications has been performed before.

Compared to the related work, we introduce metrics that describe all aspects of communication and page usage mathematically, focusing on which type of behavior makes an application suitable for mapping.

3 CHARACTERIZING COMMUNICATION BEHAVIOR



As discussed in Chapter 1, optimizing communication has the goal of improving the usage of the interconnection and caches through a better task mapping. An important aspect of communication characterization is that in general, neither applications nor the operating system perform a communication-aware task mapping. For this reason, the improvements that are expected from an improved mapping can be directly estimated from the characterization.

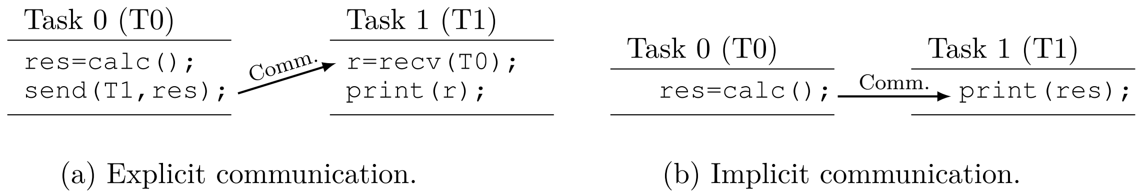
In this chapter, we discuss the qualitative and quantitative aspects of communication in shared memory architectures. For the qualitative description of communication, we define which memory accesses need to be considered as communication and how communication can be detected. Quantitatively, the communication behavior of parallel applications is then described with three properties: the *structure*, *volume*, and *temporal* components. We also present several task mapping policies that improve locality and balance of communication. In the final part of this chapter, we will analyze the parallel benchmarks using the metrics and evaluate their suitability for task mapping.

3.1 Qualitative Communication Behavior

Describing the communication behavior presents several challenges that need to be addressed. In this section, we will present definitions of communication in shared memory architectures and discuss their impact on the behavior detection, as well as the task mapping.

3.1.1 Explicit and Implicit Communication

Parallel programming models can use different forms of communication. Communication can be *explicit*, when the model uses `send()` and `receive()` functions to exchange messages between tasks, as shown in Figure 3.1a. In *implicit* communication, communication is performed directly through memory accesses to shared variables, without using explicit functions to communicate, as shown in Figure 3.1b. Explicit communication supports communication in distributed environments through message transmission over network protocols, such as TCP/IP for nodes interconnected via Ethernet. Implicit communication requires that tasks share a physical address space and is therefore limited to shared memory architectures. However, implicit communication has a lower overhead than explicit communication, since it only requires a memory access, while explicit communication has the additional overhead of the socket and packet encapsulation, among others (BUNTINAS; MERCIER; GROPP, 2006a).



(a) Explicit communication.

(b) Implicit communication.

Figure 3.1: Explicit and implicit communication between two tasks T1 and T2.

Programming APIs for explicit communication include the Message Passing Interface (MPI) (MESSAGE PASSING INTERFACE FORUM, 2012) and Charm++ (KALE; KRISHNAN, 1993), while OpenMP (DAGUM; MENON, 1998) and Pthreads (BUTTLAR; FARRELL, 1996) use implicit communication. Since communication via shared memory has a lower overhead (BUNTINAS et al., 2009), many implementations of MPI contain extensions to communicate via shared memory within cluster nodes, such as Nemesis (BUNTINAS; MERCIER; GROPP, 2006b; BUNTINAS; MERCIER; GROPP, 2006a) for MPICH2 (GROPP, 2002) and KNEM (GOGLIN; MOREAUD, 2013) for MPICH2 and Open MPI (GABRIEL et al., 2004). These extensions are usually activated by default (ARGONNE NATIONAL LABORATORY, 2014a). The extensions allocate a shared memory segment for communication and transform the MPI function calls such that they access these shared segments for communication, bypassing the network layer (BUNTINAS; MERCIER; GROPP, 2006a). For this reason, both explicit and implicit communication can be optimized by improving memory accesses in shared memory architectures.

3.1.2 True/False Communication and Communication Events

In explicit communication, all communication is *true*, that is, every call to a communication function represents an intention to exchange data between tasks. In implicit communication however, not every memory access to shared data by different tasks necessarily implies an intention to communicate. We refer to this unintentional communication as *false* communication, which can be further divided into *spatial*, *temporal* and *logical* false communication. All types of false communication are caused by the way that the hardware architecture, especially the caches and interconnections, operate. An overview of the true and false communication types is shown in Figure 3.2 for two tasks T_0 and T_1 that access the same cache line (gray box). The line consists of 4 words.

Only when two tasks access the same word in the same cache line while the line is not evicted and the second access is not an unnecessary reload, we call this access *true* communication (Figure 3.2a).

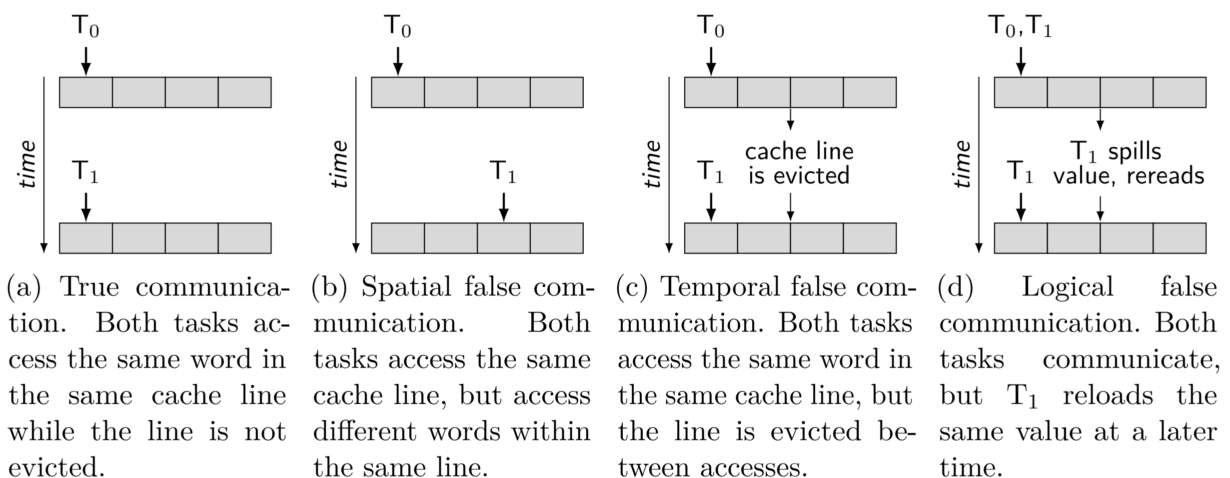
Spatial false communication happens because the granularity of cache lines and interconnections is larger than the granularity of memory accesses, similar to the classic false sharing problem (BOLOSKY; SCOTT, 1993). As an example, consider that two tasks perform a memory access to the same cache line, but at different offsets within the same line, as shown in Figure 3.2b. This access is not true communication, as it does not represent an intention to transfer data. However, the architecture treats this

access in exactly the same way as it would treat an access to the same offset, in terms of the cache coherence protocol, invalidation and transfer of cache lines. Since we are mostly interested in the architectural effects of communication, we include spatial false communication on the cache line granularity in our definition of communication. In this way, communication-aware mapping can improve accesses to truly shared data, and can reduce the impact of false sharing.

Temporal false communication happens when two tasks access the same memory address, but at different times during the execution, such that at the time of the second access, the cache line is not in the caches anymore and needs to be fetched from the main memory. This situation is shown in Figure 3.2c. This type of false communication is very dependent on the configuration and size of the caches. It can present difficulties for communication detection mechanisms that rely on memory traces and do not have a way to filter communication with a low temporal locality. Since temporal false communication affects the architectural impact of communication, we will reduce its impact by taking into account the temporal locality in our mechanisms, as described in the next chapters.

Logical false communication happens due restrictions of the hardware architecture, especially due to the limited number of registers. For example, if an application requires more registers at the same time than the hardware provides, the compiler needs to spill a register to the memory and re-read the value at a later time. Since this behavior does not constitute an exchange of data, this second access is logical false communication. However, similarly to the spatial false communication, it also affects the architecture. Therefore, we also consider these accesses as communication, in contrast to previous work that focuses on the logical communication behavior (BARROW-WILLIAMS; FENSCH; MOORE, 2009).

Summarizing the discussion, we will consider spatial and logical false communication in the same way as true communication in this thesis, and will filter temporal false communication in our mechanisms. With these considerations, we introduce the concept of a *communication event*, which we define as two memory accesses from different tasks to the same cache line while the cache line is not evicted. Some of our mechanisms will relax this definition, by increasing the granularity of the detection to a value that is larger than



(a) True communication. Both tasks access the same word in the same cache line while the line is not evicted.

(b) Spatial false communication. Both tasks access the same cache line, but access different words within the same line.

(c) Temporal false communication. Both tasks access the same word in the same cache line, but the line is evicted between accesses.

(d) Logical false communication. Both tasks communicate, but T_1 reloads the same value at a later time.

Figure 3.2: Comparison between true and false communication. Consider that two tasks T_0 and T_1 access the same cache line (gray box), which consists of 4 words.

the cache line size, and by using simpler definitions of temporal false communication that are independent of the cache configuration.

3.1.3 Read and Write Memory Accesses

Write operations are generally more expensive than reads, since they imply the invalidation of cache lines in remote caches, requiring more traffic on on-chip interconnections than the cache-to-cache transfers that are caused by read operations. However, read memory accesses are much more numerous than writes. For example, 71.1% of memory transactions in the (sequential) SPEC CPU 2006 benchmark suite (KEJARIWAL et al., 2008) are read operations, while they make up 78.1% in the PARSEC suite (BIENIA et al., 2008). Read accesses also have higher chances to stall the pipeline, since they generate more dependencies.

Moreover, the processor needs to wait for a read operation to finish in order to be able to continue operating with the just loaded cache line, which might involve waiting for the main memory. This latency can not always be hidden with Out-of-Order (OoO) execution. On the other hand, write operations are mostly asynchronous. After issuing the write, the processor only needs to wait for an acknowledgment from the L1 data cache to be able to continue with the next instruction. For these reasons, we consider both read and write memory accesses equivalently for the description of communication.

3.1.4 Communication Direction and Communication Matrix

In explicit communication, each communication operation has a well-defined sending task and a receiving task (or a group of multiple receiving tasks), in other words, communication is *directed*. In implicit communication however, determining the sender and receiver of communication is much more difficult. Three types of communication events can be defined for implicit communication, depending on whether data is read or written by two threads. These types are *read/read*, *read/write*, and *write/write*. In the *read/read* case, both tasks perform read memory accesses to the same cache line, in order to read input data for example. No task can be identified as the sender/receiver as they perform the same operation. In the *read/write* case, one task writes data which is read by the other task. In this case, the writing task can be considered the sender, and the reading task the receiver. In the *write/write* case, similar to the read/read case, sender and receiver can also not be identified. Since direction can not be determined in the majority of cases, we treat communication in shared memory as *undirected* in this thesis.

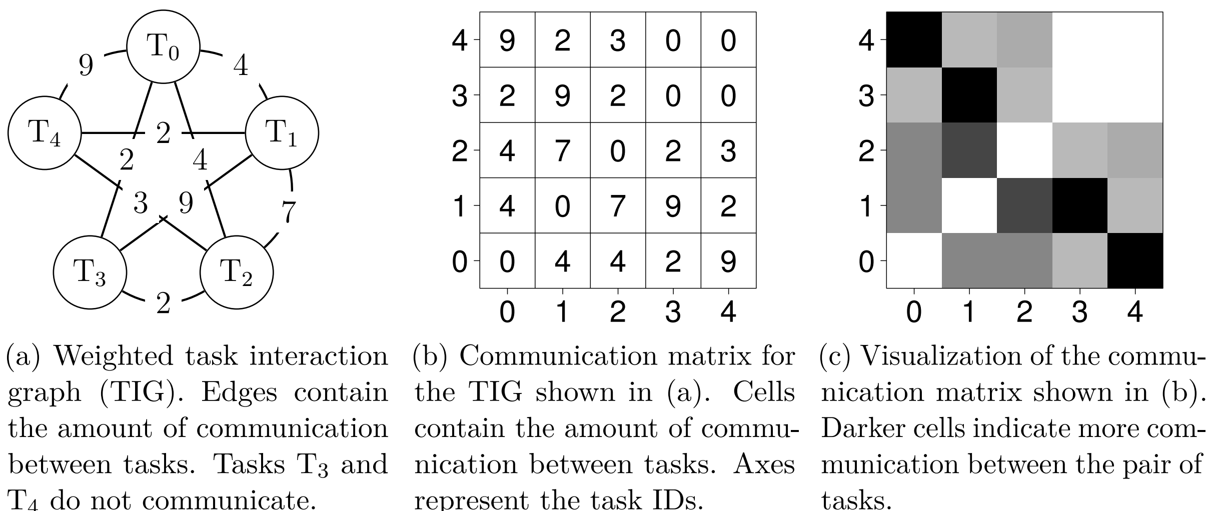
With the information about the communication events, it is possible to create an undirected communication graph, where nodes represent tasks and edges the number of communication events between each pair of tasks. An example of such a graph is shown in Figure 3.3a for a parallel application consisting of five tasks. This type of graph is also referred to as a Task Interaction Graph (TIG) in the literature (LONG; CLARKE, 1989). In practice, this communication graph is represented as a matrix, which we call *communication matrix* or *communication pattern*. An example communication matrix for

the previous graph is shown in Figure 3.3b. Each cell of the matrix contains the number of communication events for the task pairs, while the axes contain the task IDs. Since we consider that communication is undirected, the matrix is symmetric. Furthermore, the diagonal of the matrix is kept zero for the majority of the discussion in the thesis, as memory accesses by the same task do not constitute communication. Finally, to analyze and discuss the communication patterns, we generally normalize the matrices to their maximum value, to limit the range of values between 0 and 100, for example. To better visualize the communication pattern, we depict the normalized matrix in the form of a heat map, where darker cells indicate more communication. An example of this visualization is shown in Figure 3.3c.

An important aspect of communication is the question of how to compare different communication behaviors. Since a communication matrix can be thought of as a grayscale image, we use a concept from image comparison to compare different matrices. To compare two normalized communication matrices A and B that have the same size, we calculate the Mean Squared Error (MSE) (HORE; ZIOU, 2010) with Equation 3.1, where N is the number of tasks of each matrix. If A and B are equal, the MSE is equal to zero. The MSE is maximized when only a single pair of tasks communicates in one matrix and all tasks except that pair communicate equally in the other matrix. In that case, the MSE is given by $\frac{N^2-N}{N^2} \times \max(M)^2$, where $\max(M)$ is the maximum value of both matrices (the value that the matrices are normalized to).

$$MSE\ (A,B) = \frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \Big(A[i][j] - B[i][j] \Big)^2 \tag{3.1}$$

By using the MSE, it is possible to compare different communication behaviors with each other, as well as to measure the accuracy of different communication detection mechanisms.



(a) Weighted task interaction graph (TIG). Edges contain the amount of communication between tasks. Tasks T_3 and T_4 do not communicate.

(b) Communication matrix for the TIG shown in (a). Cells contain the amount of communication between tasks. Axes represent the task IDs.

(c) Visualization of the communication matrix shown in (b). Darker cells indicate more communication between the pair of tasks.

Figure 3.3: Three representations of undirected communication behavior for a parallel application consisting of five tasks, $T_0 - T_4$.

3.1.5 Relaxing the Definition of Communication

The definition of communication presented in Section 3.1.2 is the most accurate definition to analyze the architectural impact of communication, but it has three disadvantages. First, since it is based directly on the size and configuration of the cache levels, different cache configurations might result in a different communication behavior, making it less useful to describe the application behavior itself. Second, determining the communication behavior with the accurate definition requires either analyzing the application in a full cache simulator, which has a high overhead, or access to the contents of the cache on real hardware, which is not possible on most modern architectures. Third, storing and analyzing communication on the cache line granularity (64 bytes in most current architectures) has a high storage overhead due to the need to save large amounts of data. This overhead can be reduced by increasing the granularity of the analysis to large sizes than the cache line size. For these reasons, we present a relaxed definition of communication and compare it to the accurate definition in this section.

3.1.5.1 A Relaxed Definition of Communication

We relax the accurate definition of communication in the following way. First, we remove the requirement on the cache hierarchy and consider all accesses to memory addresses on a granularity derived by a common cache line size as communication events. To reduce the impact of temporal false communication, we maintain a small queue of the two most recent tasks that accessed each cache line. Second, we increase the granularity to a higher value than the cache line size, separating the memory address space into memory blocks. Algorithm 3.1 shows the function that is executed on each memory access. The memory block is calculated by bit shifting the address with the chosen granularity. The block contains a queue that stores the ID of the previously accessing tasks. Then, the number of tasks that previously accessed the block are counted. If other tasks had accessed the block before, communication events are recorded and the queue is updated.

3.1.5.2 Comparing the Communication Definitions

We compare the accurate and relaxed definitions of communication by measuring the MSE (as introduced in Section 3.1.4) of the generated communication matrices. The accurate communication is generated with a full cache simulator[1] based on the Pin dynamic binary instrumentation tool (LUK et al., 2005). The tool traces all memory accesses of a parallel application and simulates an 8-core architecture with a 2-level cache hierarchy. Each core has private L1 data and instruction caches, the L2 cache is shared among all cores. For the relaxed definition of communication, we use the Pin-based memory tracing tool presented in Section 2.3 and use Algorithm 3.1 on each memory access to determine the communication matrix. As an example of the influence of the communication definitions, we analyze the behavior of the UA-OMP benchmark from NAS-OMP with the *W* input size, which has a high sensitivity to these characteristics. All experiments were performed with 8 threads.

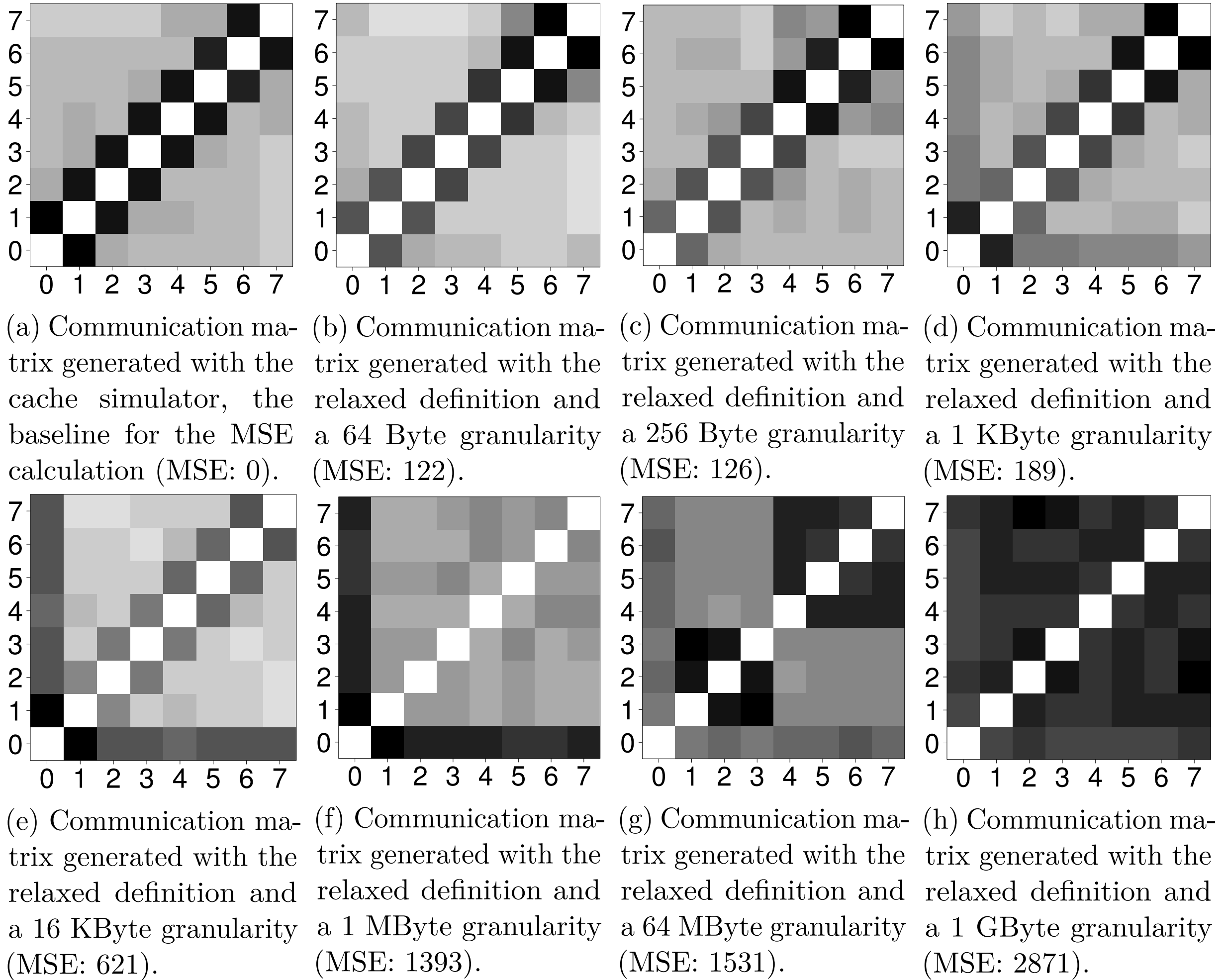
[1] CacheSim: <https://github.com/matthiasdiener/CacheSim>

Algorithm 3.1: Counting communication events with the relaxed definition.

```
   Input: address: memory address that was accessed; tid: task ID of the task that
          performed the access; granularity: granularity of detection
   // memory block of the address, contains a queue of up to 2
      tasks:
 1 block = address >> granularity;
   // number of tasks that accessed the block; can be 0, 1 or 2:
 2 nTasks = block.size();
 3 if nTasks == 0 then
       // no previous access
 4     block.push_back(tid);
 5 end
 6 if nTasks == 1 && block.front() != tid then
       // 1 previous access
 7     communication_event(block.front(), tid);
 8     block.push_back(tid);
 9 end
10 if nTasks == 2 then
       // 2 previous accesses
11     t1 = block.front();
12     t2 = block.back();
13     if t1 != tid && t2 != tid then
14         communication_event(t1, tid);
15         communication_event(t2, tid);
16         block.pop_front();
17         block.push_back(tid);
18     end
19     else if t1 == tid then
20         communication_event(t2, tid);
21     end
22     else if t2 == tid then
23         communication_event(t1, tid);
24         block.pop_front();
25         block.push_back(tid);
26     end
27 end
```

In Figure 3.4, the communication matrices of the different detection mechanisms are shown. The baseline of our evaluation, the matrix generated with the cache simulator is shown in Figure 3.4a. Figures 3.4b – 3.4h show the matrices generated with the relaxed definition and increasing granularity of memory blocks. The figure also contains the values of the Mean Square Error (MSE) as defined in Section 3.1.4, calculated between the baseline and each matrix generated with the relaxed definition. Higher MSEs indicate a higher inaccuracy of the detected communication. In this configuration with 8 threads, the maximum possible MSE is 8750.

The results show that the communication detected with the relaxed definition remains very accurate up to a granularity of 1 KByte, with low values for the MSE and matrices that are visually similar to the baseline. When increasing the granularity to



(a) Communication matrix generated with the cache simulator, the baseline for the MSE calculation (MSE: 0).

(b) Communication matrix generated with the relaxed definition and a 64 Byte granularity (MSE: 122).

(c) Communication matrix generated with the relaxed definition and a 256 Byte granularity (MSE: 126).

(d) Communication matrix generated with the relaxed definition and a 1 KByte granularity (MSE: 189).

(e) Communication matrix generated with the relaxed definition and a 16 KByte granularity (MSE: 621).

(f) Communication matrix generated with the relaxed definition and a 1 MByte granularity (MSE: 1393).

(g) Communication matrix generated with the relaxed definition and a 64 MByte granularity (MSE: 1531).

(h) Communication matrix generated with the relaxed definition and a 1 GByte granularity (MSE: 2871).

Figure 3.4: Comparison of the communication matrices of the UA-OMP benchmark, generated with the accurate (a) and relaxed (b) – (h) definitions of communication with different detection granularities. The MSE is calculated as the difference to the accurate matrix (a).

values above 1 KByte, the MSE keeps rising and the matrices lose their similarity to the baseline, with a complete divergence starting at about 1 MByte. These results indicate that the relaxed definition is accurate enough to be used to analyze communication if the granularity of detection is less than 1 Kbyte. Since UA-OMP has a very low memory usage (8 MByte for the configuration shown here), the accuracy presented here is a lower bound. Applications with a higher memory usage have a lower sensitivity to the granularity since there is less overlap between the memory blocks.

As expected, generating the communication with the relaxed definition is much faster than the cache simulator, about 20× faster for this benchmark, with even higher speedups in general for larger benchmarks (about 35× for UA-OMP with input size A, for example). For this reason, we will use the relaxed definition with a 64 Byte granularity (which is the cache line size of most current architectures) for our evaluation of the benchmark behavior in this chapter, since it allows us to analyze even huge parallel applications with a high accuracy. For the automatic mapping mechanisms that will be

presented in Part II of this thesis, we will use the relaxed definition with higher granularities to allow a more efficient analysis of runtime behavior.

3.1.6 Common Types of Communication Patterns

Communication patterns are the most important aspect of communication, since they determine the task mapping that is calculated. Many parallel applications have one of several common types of communication patterns. This section will present and discuss the most common patterns: initialization, reduction, all-to-all, nearest neighbor, matrix, and pipeline. Code examples[2] that generate each pattern, written in OpenMP, as well as the patterns themselves are shown in Figure 3.5. The examples are shown for an application consisting of 8 tasks. The patterns were verified with gcc 4.6, but naturally depend on how the compiler and its runtime library manages tasks, such that different compilers might produce different patterns.

In an *initialization* pattern, shown in Figure 3.5a, a task distributes initial input data to the other tasks. Usually, the first created task, task 0, performs this distribution. This behavior results in a communication pattern similar to the one that is shown in the Figure, where task 0 performs a lot of communication with all other tasks, but the other tasks only communicate very little among themselves.

In a *reduction* pattern, a task collects partial results from other tasks and performs a reduction operation, such as a sum, on the data. The pattern, shown in Figure 3.5b, is very similar to the initialization pattern. In the example, task 0 collects information from all other tasks.

In an *all-to-all* communication pattern, all tasks communicate in an equal way with each other, such that all pairs of tasks have a similar amount of communication. This situation is depicted in Figure 3.5c. The code example performs a matrix multiplication. Since the values of the resulting matrix depend on values from all rows and columns of the input matrices, communication between all tasks is performed.

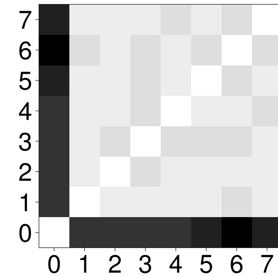
In a *nearest neighbor* pattern, most communication happens between neighboring tasks, as shown in Figure 3.5d, where task pairs (0,1), (1,2), ..., communicate the most. This behavior is common in applications that are based on domain decomposition, where communication happens via shared data on the borders of the domains. The code presented in the figure performs the Gauss–Seidel method to solve a linear system of equations.

The *matrix* communication pattern, shown in Figure 3.5e, is similar to the nearest neighbor pattern. However, in addition to lots of communication between directly neighboring tasks, there is also substantial communication between tasks that are farther apart, such as tasks 0 and 6 in the example. This behavior is common in algorithms that operate on cells of matrices, where the new cell depends on values in neighboring cells in the same row (which are also accessed by neighboring tasks), as well neighboring cells in different rows (which are also accessed by tasks that are farther apart). The code presented in the figure calculates a Gaussian blur for a matrix.

---

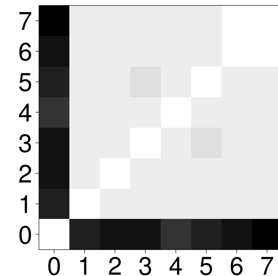
[2] The examples shown in the figures only contain parts of the main function due to space constraints. The full code is available at <https://github.com/matthiasdiener/communication-samples>.

```
1 int a[N];
2 for (i=0; i<N; i++)
3   a[i] = i;
4
5 #pragma omp parallel
6 {
7   int sum, i, tid = omp_get_thread_num();
8   for (i=tid*N/8; i<(tid+1)*N/8; i++)
9     sum += a[i];
10 }
```



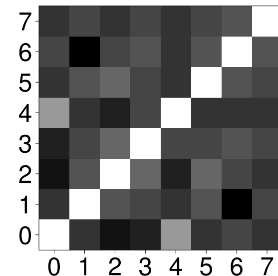
(a) Initialization communication pattern.

```
1 int i;
2 int a[N];
3 int sum = 0;
4
5 #pragma omp parallel for reduction(+:sum)
6 for (i=0; i < N; i++) {
7   sum += a[i];
8 }
```



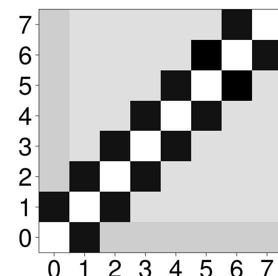
(b) Reduction communication pattern.

```
1 int A[N][N], B[N][N], C[N][N], i, j, k;
2
3 #pragma omp parallel for
4 for(i=0; i<N; i++) {
5   for(j=0; j<N; j++) {
6     for(k=0; k<N; k++) {
7       C[i][j] += A[i][k]*B[k][j];
8     }
9   }
10 }
```



(c) All-to-all communication pattern of a naive matrix multiplication.

```
1 #pragma omp parallel
2   {
3     jstart = tid*dnp; jstop = jstart + dnp;
4     for(j=jstart; j<jstop; j++)
5       dxi += A[i][j]*x[j];
6 #pragma omp critical
7       dx[i] -= dxi;
8   }
9   dx[i] /= A[i][i]; x[i] += dx[i];
10  sum += abs(dx[i]);
```

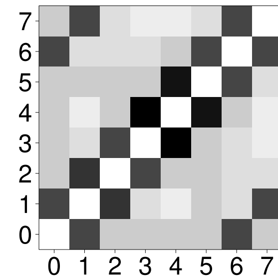


(d) Nearest neighbor communication pattern of the Gauss-Seidel method. Adapted from (VERSCHELDE, 2014).

Figure 3.5: Common types of communication patterns.

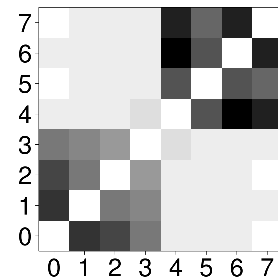
In applications with a *pipeline* pattern, communication happens usually within each pipeline stage, but there is little communication between different stages. An example of this pattern is shown in Figure 3.5f. In the example, there are 2 pipeline stages consisting of

```
1 #pragma omp parallel for
2 for (y = 0; y < h; y++) {
3   int tid = omp_get_thread_num(), x1;
4   for (x1 = 0; x1 < w; x1++) {
5     if(x1 < xmax)
6       ringbuf[ksize*tid] = src[y*w + x1+halfk];
7     else
8       ringbuf[ksize*tid] = src[y*w + w-1];
9     dst[y*w + x1] = convolve(kernel,
        &ringbuf[ksize*tid], ksize, bufi0);
10  }
11 }
```



(e) Matrix communication pattern of a Gaussian blur.

```
1 int A[N], sum, flag=0, i;
2 #pragma omp parallel
3 {
4   int tid = omp_get_thread_num();
5   if (tid<4) {
6     for (i=tid*N/8;i<(tid+1)*N/8;i++)
7       A[i] = random(); //produce
8     flag = 1;
9   } else {
10    while (flag != 1) {};
11    for (i=tid*N/8;i<(tid+1)*N/8;i++)
12      sum += A[i]; //consume
13  }
14 }
```



(f) Pipeline communication pattern of a producer-consumer benchmark. Adapted from (MATTSON; MEADOWS, 2008).

Figure 3.5: Common types of communication patterns (ctd.).

4 tasks each, with large amounts of communication in each stage (0–3 and 4–7). The code shown in the figure is a producer-consumer application, where the first 4 tasks produce random data in the `A` vector, which is then consumed by the other 4 tasks. In addition to the communication within each pipeline stage, the shared data is communicated between the stages as well.

Applications can also show a combination of several patterns. For example, an application might be distributing input data during initialization, then communicate with a nearest neighbor pattern, and finally collect partial results with a reduction.

With the patterns presented here, it is already possible to determine informally which types of patterns can be improved with an improved task mapping (this will be discussed in detail in Section 3.2). The initialization, reduction, and all-to-all patterns are less suitable for an improved mapping, since grouping tasks according to the communication between them will always result in a similar amount of communication in each group. As an example, consider that we want to create an optimized mapping for an all-to-all pattern in an architecture that has 2 caches that are shared between 4 cores each. Any

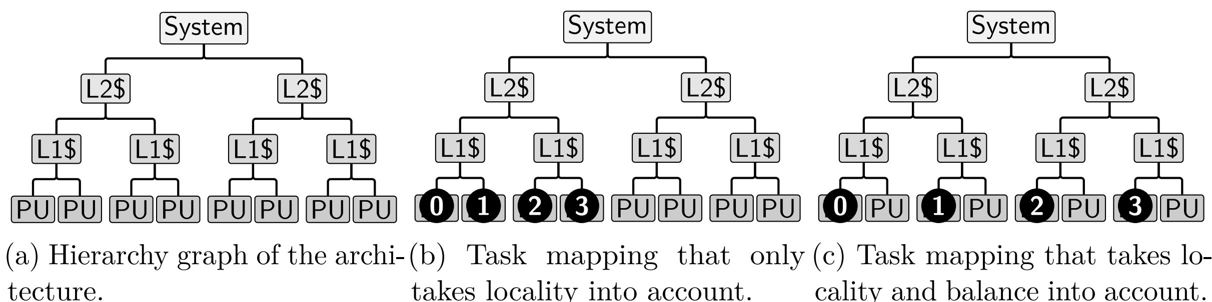
mapping of the 8 tasks to the 8 cores will result in the same amount of communication in each cache and between the caches.

On the other hand, the nearest neighbor, matrix and pipeline patterns are more suitable for mapping, since groups of processes with lots of communication within the group and little communication to other groups can be found. As an example, consider that we want to create an optimized mapping for a nearest neighbor pattern for the same architecture described before. In this case, there is an optimized mapping, by placing tasks 0–3 on one cache and tasks 4–7 on the other cache. In this way, there is lots of communication in each cache, but very little communication between the caches.

3.1.7 Communication Balance

Apart from the *locality* of communication, another important metric to consider is the *balance* of communication. To illustrate the importance of balance, consider an the architecture shown in Figure 3.6a, which consists of 8 processing units (PUs), 2 PUs sharing an L1 cache, and 4 PUs sharing an L2 cache. Consider also that we want to create an optimized task mapping of 4 tasks with a nearest neighbor communication pattern to this architecture. In this scenario, a locality-based mapping policy might create a mapping similar to the one shown in Figure 3.6b, which maximizes local communication through 2 L1 caches and 1 L2 cache. Although this mapping is optimal in terms of locality, it is severely imbalanced, as half of the caches are not used at all. Therefore, a mapping that also takes into account the communication balance as well might result in a higher overall performance. Such a mapping is shown in Figure 3.6c, which uses all the caches, while still taking the locality into account.

Balance needs to be taken into account for two reasons. First, in situations where there is an overprovisioning or underprovisioning of the hardware, that is, when there are more or fewer tasks than the architecture can execute in parallel. The example in the first paragraph is an underprovisioning scenario. A second scenario is when the communication of the tasks is imbalanced, that is, when some tasks have a higher or lower amount of communication than others. An example of this situation is shown in Figure 3.7, showing communication matrices for balanced communication (Figure 3.7a) and imbalanced communication (Figure 3.7b). To discuss the communication balance,

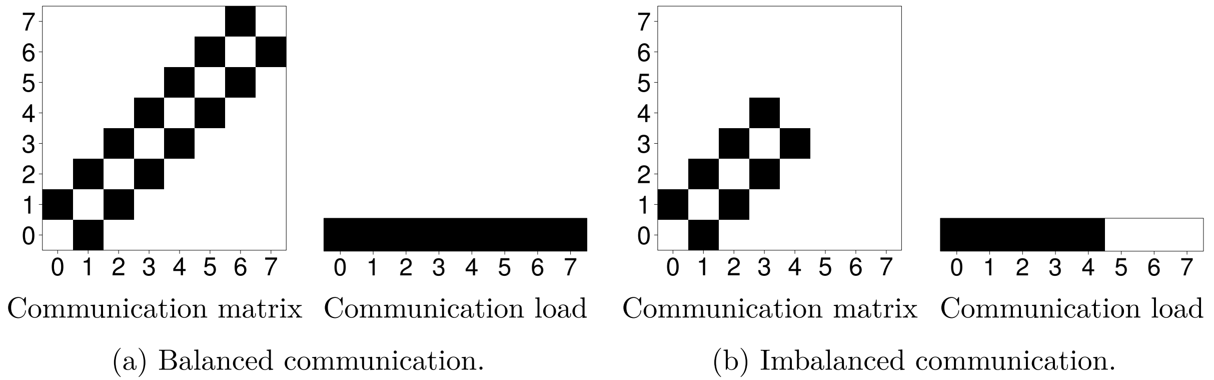


(a) Hierarchy graph of the architecture.

(b) Task mapping that only takes locality into account.

(c) Task mapping that takes locality and balance into account.

Figure 3.6: Locality and balance of communication in an architecture with 8 PUs and 4 tasks with a nearest neighbor communication pattern.



(a) Balanced communication.

(b) Imbalanced communication.

Figure 3.7: Communication balance.

we linearize the communication matrix by summing up all columns, generating the total amount of communication per task. We visualize communication balance in a balance vector, where darker cells indicate more communication by a task. The vectors show that in the first case, communication is perfectly balanced, while in the second case, there is a large imbalance due to the fact that tasks 5–7 do not communicate. If none of these two situations exists, that is, if the number of tasks is equal to the number of PUs and all tasks perform the same amount of communication, no special balancing has to be performed.

Communication imbalance of a parallel application can happen for several reasons. For example, if the problem to be solved can not be divided equally for the requested number of tasks, some tasks will work with a smaller (or even empty) part of the problem, leading to a reduced amount of communication for these tasks. Another reason for imbalance is if some parts of the problem inherently require more communication to be solved. This can happen in a pipeline parallelization, for example, where some stages might require more communication than others. Not all communication imbalance can be solved with an improved mapping. The imbalance of the examples shown in Figures 3.6 and 3.7 can be solved by a balanced mapping. However, when only a few tasks have a significant amount of communication (such as in the initialization or reduction patterns), balancing the communication is much more difficult, as the caches to which the tasks with most communication get mapped will always handle more communication than the other caches.

Communication balance is a complement to load balance. In load balance, the amount of computation performed by each task is referred to as the *load* of that task. The goal of a load balancing algorithm is to distribute the load such that all processing units perform the same amount of computation. As mentioned before, tasks might have different computational and communication requirements, which can lead to higher improvements from communication balance than from load balance, depending on these requirements. Due to the similarity, we reuse some concepts of load balance in this chapter.

3.2 Quantitative Communication Behavior

After presenting the qualitative aspects of communication in shared memory, we will formalize the concepts discussed in Section 3.1 to derive quantitative descriptions of

the behavior. The quantitative communication behavior of parallel applications can be described with three properties: the *structure*, *volume*, and *temporal* components (KIM; LILJA, 1998; SINGH; ROTHBERG; GUPTA, 1994). These metrics are used to determine which communication behavior can benefit from task mapping policies that are based on locality, balance, or both. In this section, we will first introduce metrics for the structure of communication, which we call *heterogeneity* and *balance*, and metrics for the volume of communication, which we call the *amount* and *ratio* of communication. Then, we will describe the temporal behavior in terms of changes to these metrics during the execution of the parallel application.

3.2.1 Communication Heterogeneity

For policies that focus on improving the locality of communication, it is necessary to have groups of threads that communicate more within the group than with threads outside the group. Conversely, if all tasks communicate in the same way among them, no improvements from locality-based task mapping can be expected. To formalize this intuition, we can determine the potential for task mapping of the application by analyzing the differences between the amounts of communication between the tasks contained in the communication matrix.

We describe the differences as the *heterogeneity* of the communication C_H. To calculate it, we first normalize the communication matrix to its maximum value, as shown in Equation 3.2, where $\max(M)$ returns the maximum value of M. Normalization makes it easier to compare the heterogeneity between different applications. Due to the symmetry of the matrix, we use only the upper triangular. Then, the heterogeneity C_H is calculated with Equation 3.3, where T is the number of tasks, the var function calculates the variance and then min function returns the minimum value of its arguments. The equation calculates the average variance of the communication of each task in both x and y-direction and chooses the minimum of these two values. The higher the heterogeneity, the higher the potential for locality-based task mapping, since there are larger differences between the amounts of communication. If the heterogeneity is low, the amount of communication between the tasks is similar, thus any task mapping applied would result in a similar locality of communication.

$$M_{norm} = M / \max(M) \times 100 \tag{3.2}$$

$$C_H = \frac{\min\left(\left[\sum_{i=1}^{T} \mathrm{var}(M_{norm}^{upper}[i][1...T])\right], \left[\sum_{i=1}^{T} \mathrm{var}(M_{norm}^{upper}[1...T][i])\right]\right)}{T} \tag{3.3}$$

3.2.2 Communication Balance

For task mapping policies that are based on balance, it is necessary to determine if some threads are performing more communication than others. To evaluate this property,

we introduce the C_B metric, which we refer to as the *balance* of the threads' communication behavior. To calculate C_B, we first calculate the total amount of communication per thread in a *communication vector* CV, where each element i of CV contains the number of communication events of thread i. The vector can be directly generated from the communication matrix, by summing up the communication events per row. As the communication matrix is symmetric, we only use the upper half of the matrix (M_{upper}) for the calculation. The CV is generated with Equation 3.4, where rowSums calculates the sum for each row of the matrix.

$$CV = \text{rowSums}(M_{upper}) \tag{3.4}$$

Similar to the communication matrix, the CV is then normalized to values between 0 and 100 with Equation 3.5.

$$CV_{norm} = CV/\max(CV) \times 100 \tag{3.5}$$

With the normalized CV, we calculate the communication imbalance C_B with Equation 3.6, similar to traditional load imbalance (PEARCE et al., 2012).

$$C_B = \left(\frac{\max(CV_{norm})}{\sum_{i=1}^{T} CV_{norm}[i]/T} - 1 \right) \times 100\% \tag{3.6}$$

A value of 0 for C_B indicates that the communication between tasks is perfectly balanced, while higher values indicate an imbalance in the communication behavior, suggesting that balance-based task mapping policies can be beneficial. Figure 3.8 shows a comparison of communication matrices with different values of heterogeneity and balance.

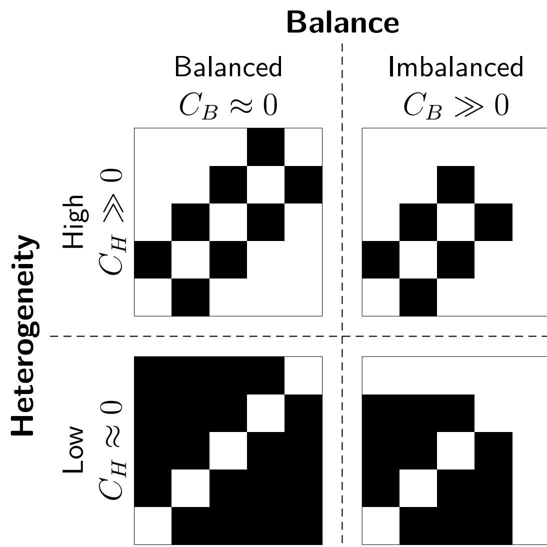


Figure 3.8: Comparison of the structure of communication with matrices that have different values for heterogeneity and balance, for an application consisting of five tasks.

3.2.3 Communication Amount

Improvements according to a specific task mapping policy also depend on how much tasks are communicating. We expect higher gains for parallel applications that communicate more. To describe the *amount* of communication C_A, we use the average number of communication events per task and per second of execution time, calculated with Equation 3.7. When there is little communication, task mapping has less potential for improvements, even when the heterogeneity or imbalance are high.

$$C_A = \frac{\sum_{i=1}^{T} \sum_{j=1}^{T} M[i][j]}{T^2 \times \mathit{execution\ time}} \tag{3.7}$$

3.2.4 Communication Ratio

The amount of communication itself is not sufficient to evaluate if an application is suitable for communication-aware task mapping. If tasks have much more memory accesses to private data than communication, a communication-aware mapping might not affect the overall memory access behavior. For this reason, we define the *communication ratio* metric C_R, which is the ratio of the communication accesses to the total number of memory accesses of the application tasks. C_R is calculated by Equation 3.8, where $AccV[i]$ is the number of memory accesses performed by task i.

$$C_R = \frac{C_A}{\sum_{i=1}^{T} AccV[i]} \tag{3.8}$$

3.2.5 Temporal Communication Behavior and Phase Detection

The previous two metrics describe the state of the communication matrix at a certain point during execution. To analyze the temporal communication behavior, we separate the execution into time slices of a fixed length and calculate the heterogeneity and amount of communication for each slice.

Since the values for consecutive time slices may differ greatly, we perform a *phase detection* on the temporal behavior of the application using the heterogeneity value of the time slices. We take only the heterogeneity into account, since it determines the need to migrate tasks. A new phase starts when the average heterogeneity of the last n time slices differs from the n time slices before them by at least diff_{min}. This intuition is formalized in Equation 3.9, where t is the current time slice, $(x : y)$ represents the time slice range from x to y, and $\mathrm{havg}(t)$ represents the average heterogeneity of time slice t.

$$\frac{|\,\mathrm{havg}(t-n:t) - \mathrm{havg}(t-2n:t-n)|}{\mathrm{havg}(t-n:t)} \geq \mathit{diff}_{min} \tag{3.9}$$

In this thesis, we use a time slice length of 10ms and $n = 50$, which leads to a minimum phase length of 500ms. We use a value of 0.2 for diff_{min}. It is important to

Table 3.1: Overview of the communication metrics introduced in this section.

Metric	Symbol	Describes	Task mapping	Suitability
Heterogeneity	C_H	Structure of	Locality-based	High
Balance	C_B	communication	Balance-based	Low
Amount	C_A	Volume of	Both	High
Ratio	C_R	communication		High
Dynamicity	C_D	Temporal comm.	Both	Low

mention that these values are used to analyze and compare the dynamic behavior of the benchmarks, and do not necessarily indicate that tasks need to be migrated.

By determining the number of phases detected by Equation 3.9, we then calculate the *dynamicity* C_D of communication by normalizing it to the execution time, as shown in Equation 3.10. We expect higher improvements from task mapping from applications with a lower dynamicity, since task migrations during execution impose an overhead on the application and therefore impact the total execution time.

$$C_D = \frac{\#detected\ phases}{execution\ time} \tag{3.10}$$

3.2.6 Summary of Communication Metrics

Table 3.1 contains a summary of the communication metrics that were introduced in this section, showing which component of the communication behavior they describe, which type of task mapping they are more suitable for, and if a high or low value of the metric indicates higher suitability for the task mapping.

3.3 Task Mapping Policies

Several task mapping policies that optimize different characteristics will be evaluated: *Operating System (OS)*, *Compact*, *Scatter*, *Locality*, *Distance*, *Balance*, and *Balanced Locality*. OS, Compact, Scatter, and Locality are traditional mapping policies. The other three policies are introduced in this thesis.

3.3.1 The Policies

3.3.1.1 Operating System (OS)

The mapping performed by the operating system represents the baseline for our task mapping policies. We use the Completely Fair Scheduler (CFS) (WONG et al., 2008) of the Linux kernel, which is the default thread scheduler since version 2.6.23. The scheduler focuses mostly on fairness and load balance (WONG et al., 2008), and has no means for improving communication locality or balance.

3.3.1.2 Compact

The Compact mapping performs a round-robin scheduling of threads to PUs such that neighboring threads are placed close to each other in the memory hierarchy. This mapping can increase the locality of communication behaviors where neighboring threads communicate frequently with each other. Compact is similar to task mapping policies that are available in some MPI runtime environments (ARGONNE NATIONAL LABORATORY, 2014b) and OpenMP libraries (INTEL, 2012b).

3.3.1.3 Scatter

The Scatter policy represents the opposite of Compact, and is also available in some runtime environments (INTEL, 2012b). In this mapping, neighboring threads are placed far from each other in the hierarchy. In this way, performance can be improved for applications with little communication or a low communication ratio, by reducing competition for cache space. Compact and Scatter do not take the actual communication behavior into account.

3.3.1.4 Locality

The Locality policy optimizes the communication behavior by mapping threads that communicate frequently close to each other in the memory hierarchy. The mapping algorithm receives as input the communication matrix and a description of the memory hierarchy of the system, generated with `hwloc` (BROQUEDIS et al., 2010b). It outputs a thread mapping that maximizes the overall locality of communication. Since locality-based task mapping has received a lot of attention in recent years, many algorithms have been proposed to calculate such a mapping. Proposing a new algorithm is not part of this thesis, but we present here a brief overview of previous proposals and discuss our selected algorithm that will be used for all Locality policies throughout this thesis.

The task mapping problem is NP-hard (BOKHARI, 1981), therefore it is necessary to use efficient heuristic algorithms to calculate the mapping. The mapping problem can be modeled with two undirected graphs, a communication graph and a hierarchy graph. In the communication graph, vertices represent tasks and edges represent the amount of communication between them. In the hierarchy graph, vertices represent levels of the memory hierarchy and PUs, while edges represent the interconnections between them. The communication graph is obtained from the communication matrix, while the hierarchy graph is generated from hardware information.

Many algorithms for locality-based task mapping are based on graph partitioning, such as METIS (KARYPIS; KUMAR, 1998), ParMETIS (KARYPIS; KUMAR, 1996), Zoltan (DEVINE et al., 2006), Scotch (PELLEGRINI, 1994) and the topology framework for MPI/SX (TRÄFF, 2002). A comprehensive overview and comparison of graph-based algorithms is shown by Glantz et al. (2015). However, many shared-memory hardware topologies are hierarchical and may be represented as trees (TRÄFF, 2002), which can lead to more efficient algorithms. Examples of tree based mapping algorithm

are TreeMatch (JEANNOT; MERCIER, 2010; JEANNOT; MERCIER; TESSIER, 2014) and EagerMap (CRUZ et al., 2015).

Since our main proposals focus on online mapping during execution, there are several important criteria for choosing a good algorithm: (1) scalability, to allow mapping for large architectures, (2) execution time, to reduce the overhead on the running application, (3) mapping quality, to increase the benefits from task mapping, and (4) stability, to make sure that small changes in the communication behavior only result in small changes to the mapping in order to reduce the number of unnecessary task migrations during execution. An extensive comparison (CRUZ et al., 2015) showed that the EagerMap algorithm has the best properties for these criteria. It scales with $\mathcal{O}(T^3)$, where T is the number of tasks, which is equal to most algorithms, but has a very short execution time (up to 10× faster than Scotch, the second fastest), while maintaining the same quality as the other algorithms. Due to its greedy strategy, it has a very high stability, and results in fewer task migrations due to small changes in the communication pattern than other algorithms. For these reasons, we selected EagerMap as the task mapping algorithm for all experiments in this thesis.

3.3.1.5 Distance

The Distance policy represents the opposite of Locality, placing threads that communicate far apart in the memory hierarchy. We calculate this mapping by inverting the communication matrix, subtracting each cell by the maximum value of the matrix while keeping the diagonal of the matrix at zero. An example of such an inverted matrix is shown in Section 3.3.2. We then apply the same mapping algorithm as for the Locality mapping to the inverted matrix. This mapping can be useful when the heterogeneity is high, but the communication ratio is low, similar to the Scatter policy, but taking the actual communication behavior into account.

3.3.1.6 Balance

The Balance policy focuses on maximizing the communication balance for the application. The mapping algorithm receives the communication vector (introduced in Section 3.2.2) and the description of the memory hierarchy as input. The mapping is calculated by selecting the thread with the highest amount of communication that has not been mapped to a PU yet. This thread is then mapped to the PU which currently has the lowest amount of communication mapped to it. This process is repeated until all threads are mapped to a PU. This policy focuses only on balance and does not take locality into account.

3.3.1.7 Balanced Locality

The Balanced Locality policy focuses on increasing locality while still maintaining the balance of the communication. First, it maps threads that communicate frequently to nearby PUs, similar to the Locality policy. Second, for each level of the memory hierarchy, it keeps a similar amount of communication for each cache memory of that

level. We model the memory hierarchy as a tree, where the leaves represent the PUs, and the other levels of the tree represent cache levels and their nodes represent specific cache memories. Our algorithm groups threads with high amounts of communication to the leaves of the tree, propagating this mapping to the parent nodes up to the root node. We add threads to the leaves until the amount of communication is higher than the average amount of communication per leaf. Summarizing, this policy maps threads that communicate frequently to close PUs whose amounts of communication are lower than the average amount of communication per PU. Balanced Locality is based on an extension of EagerMap.

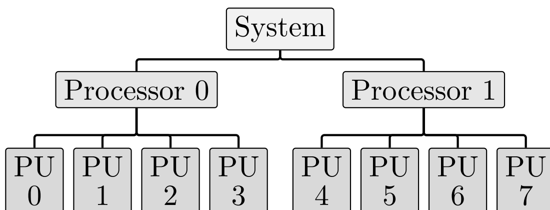
3.3.1.8 Load Balance

We also evaluate the Load Balance of selected benchmarks to compare it to the *Communication Balance* metric introduced in Section 3.2.2. We use the number of executed instructions per thread as the metric for the load, and calculate its value with the same algorithm as for the Communication Balance.

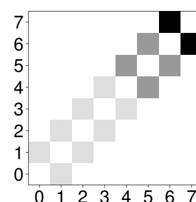
3.3.2 Example of Policy Behavior

To illustrate the various task mapping algorithms, consider the example presented in Figure 3.9. In the example, we use an architecture with two processors (4 PUs each), with 8 PUs in total (Figure 3.9a). PUs 0–3 belong to the first processor, while PUs 4–7 belong to the second one. For the example, we assume that each processor has only a single cache shared by all PUs, such that communication within each processor is faster than communication between processors. Communication performance within each processor is homogeneous.

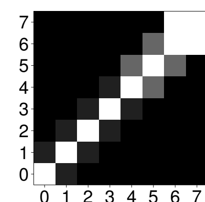
This architecture executes an application consisting of 8 tasks with the communication pattern shown in Figure 3.9b. The pattern shows communication between neighboring tasks, where one task pair (6,7) has much more communication compared to the other tasks. This pattern is very similar to the communication matrix of UA-OMP that will be presented in Section 3.4.1.1 and illustrates the behavior of the different mapping policies well. Its inverted pattern, used for the calculation of the Distance mapping, is shown in Figure 3.9c.



(a) System with 2 processors and 4 PUs each. Each processor has only a single cache.



(b) Communication pattern with 8 tasks.



(c) Inverted communication pattern of (b).

Figure 3.9: Inputs for the task mapping example.

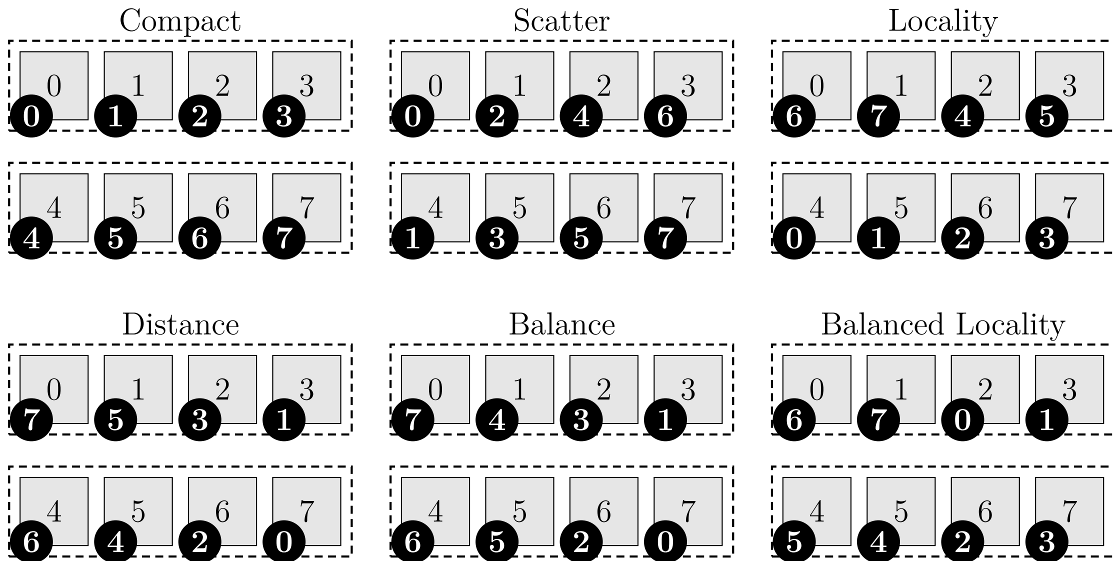


Figure 3.10: Output of the task mapping example with various algorithms. Gray boxes represent the PUs, while black circles represent the tasks.

The mappings that were calculated by each algorithm are shown in Figure 3.10. Each gray box represents one of the PUs, while the black circles indicate where each task is mapped. The Compact mapping results in a very imbalanced mapping, since processor 1 performs most of the communication. It shows a high locality though, since tasks 5–7, which communicate the most, are located in the same processor. The Scatter mapping shows a better balance, but reduces the locality of communication, since task pairs that communicate are placed on different processors. For example, tasks 6 and 7, which communicate a lot, are mapped to different processors.

The Locality policy calculated a mapping that maximizes the locality, similar to the Compact mapping, but also causes a significant imbalance. The Distance policy results in a minimal locality, as expected, but has a high communication balance. Similarly, the Balance policy results in a high balance, but also causes a lot of inter-processor communication. Finally, Balanced Locality shows a tradeoff between locality and balance. Most task pairs with lots of communication between them, (6,7) and (4,5), are placed on the same processor (causing a high communication locality), while still maintaining a good balance between processors.

3.4 Communication Behavior of the Benchmarks

For the evaluation of the in this section, we execute the parallel applications discussed in Section 2.1 with 64 tasks on a simulated machine with 64 PUs and trace all memory accesses of the application on a per-task basis. We use the *numalize* memory access tracer described in Section 2.3 with the relaxed definition of communication introduced in Section 3.1.5. Benchmarks that can not be executed with 64 tasks are executed with a number of tasks as close as possible to 64. The NAS-MZ benchmarks were executed with 4 processes and 16 threads per process. To estimate the accuracy of the detected patterns,

we performed a visual comparison of communication matrices with related work (ZHAI; SHENG; HE, 2011; MA et al., 2009; RIESEN, 2006), and did not notice any significant difference to the patterns detected by numalize. Please also see our discussion of numalize's accuracy compared to a cache simulator in Section 3.1.5. We begin with a discussion of the global communication behavior and then analyze the dynamic behavior during execution.

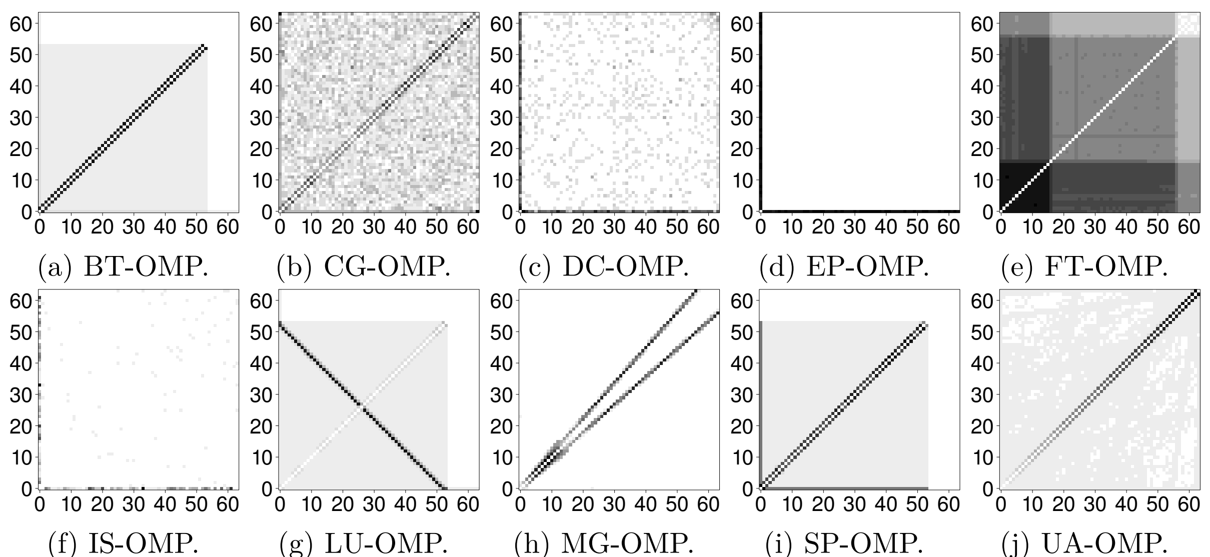
3.4.1 Global Communication Behavior

This section presents the communication patterns during the full execution of each application, to analyze their overall suitability for different task mapping policies.

3.4.1.1 NAS-OMP Benchmarks

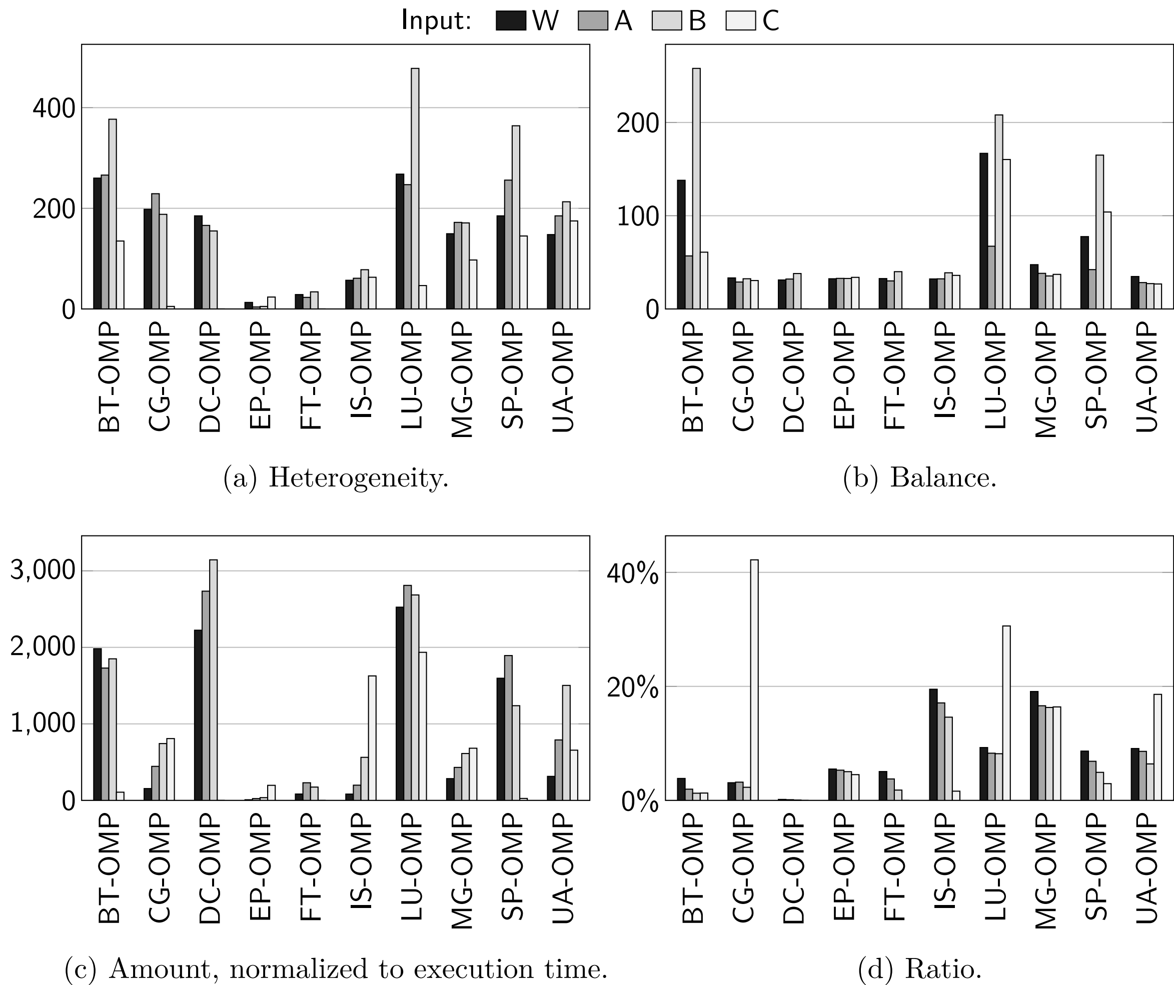
Figure 3.11 shows the communication matrices of the NAS-OMP benchmarks for the *B* input size, while Figure 3.12 shows the values of the communication metrics. Evaluating the *heterogeneity* of communication, several benchmarks, such as BT-OMP, MG-OMP, and SP-OMP, show large amounts of communication between neighboring threads, such as threads 0 and 1. SP-OMP also shows significant communication between thread 0 and all other threads, indicating an initialization and reduction behavior. LU-OMP has a high heterogeneity as well, but communication happens mostly between distant threads, such as threads 0 and 53. It is important to mention that for BT-OMP, LU-OMP, and SP-OMP, the threads with the highest IDs communicate only very little. This indicates that the communication is imbalanced. DC-OMP, EP-OMP, FT-OMP, and IS-OMP are applications with a low heterogeneity, demonstrated by their homogeneous communication matrices.

Evaluating the communication *balance* shows that only BT-OMP, LU-OMP, and SP-OMP are significantly imbalanced and show a suitability for balance-based policies. The reason for the imbalance of these applications is shown in Figure 3.11, as some of



(a) BT-OMP. (b) CG-OMP. (c) DC-OMP. (d) EP-OMP. (e) FT-OMP.
(f) IS-OMP. (g) LU-OMP. (h) MG-OMP. (i) SP-OMP. (j) UA-OMP.

Figure 3.11: Communication matrices of the NAS-OMP benchmarks (*B* input).



(a) Heterogeneity.

(b) Balance.

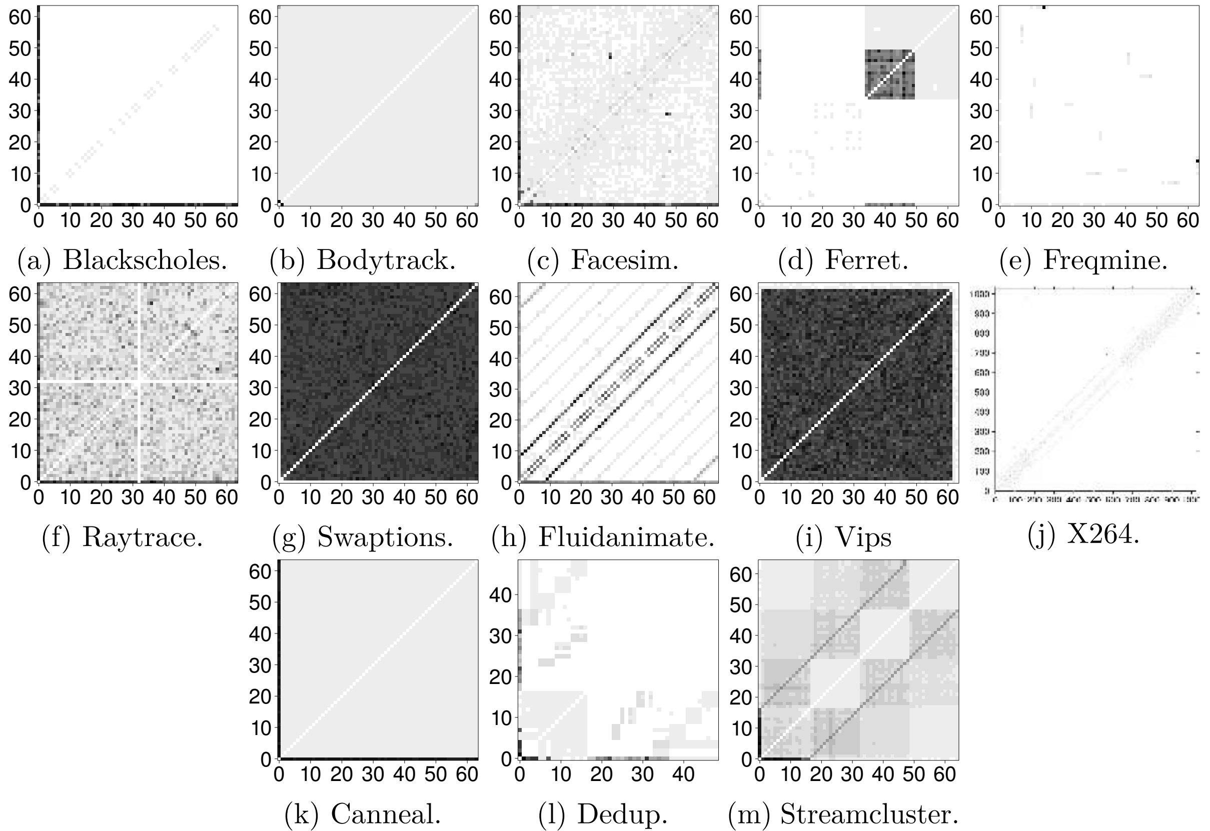
(c) Amount, normalized to execution time.

(d) Ratio.

Figure 3.12: Communication characteristics of the NAS-OMP benchmarks.

the threads are not communicating at all. This behavior changes with the input size: inputs W and B are imbalanced, while A is much more balanced. However, despite this communication imbalance, there is no significant load imbalance for these benchmarks according to our measurements, showing that the threads that communicate less still perform substantial amounts of computation. For example, SP-OMP with the B input has a load balance of only 38.1 (calculated with the number of executed instructions per thread), while the communication balance metric is much higher (165.5, higher values indicate a higher imbalance).

Analyzing the *volume* of communication metrics, we can see that the communication *amount* increases slightly with larger input sizes for most NAS-OMP benchmarks. However, the communication *ratio* shows that for most benchmarks, with increasing input sizes, less communication in comparison to the total number of memory accesses is performed. This indicates that larger input sizes of NAS-OMP are less suitable for communication-aware task mapping in general.



(a) Blackscholes. (b) Bodytrack. (c) Facesim. (d) Ferret. (e) Freqmine. (f) Raytrace. (g) Swaptions. (h) Fluidanimate. (i) Vips (j) X264. (k) Canneal. (l) Dedup. (m) Streamcluster.

Figure 3.13: Communication matrices of the PARSEC benchmarks.

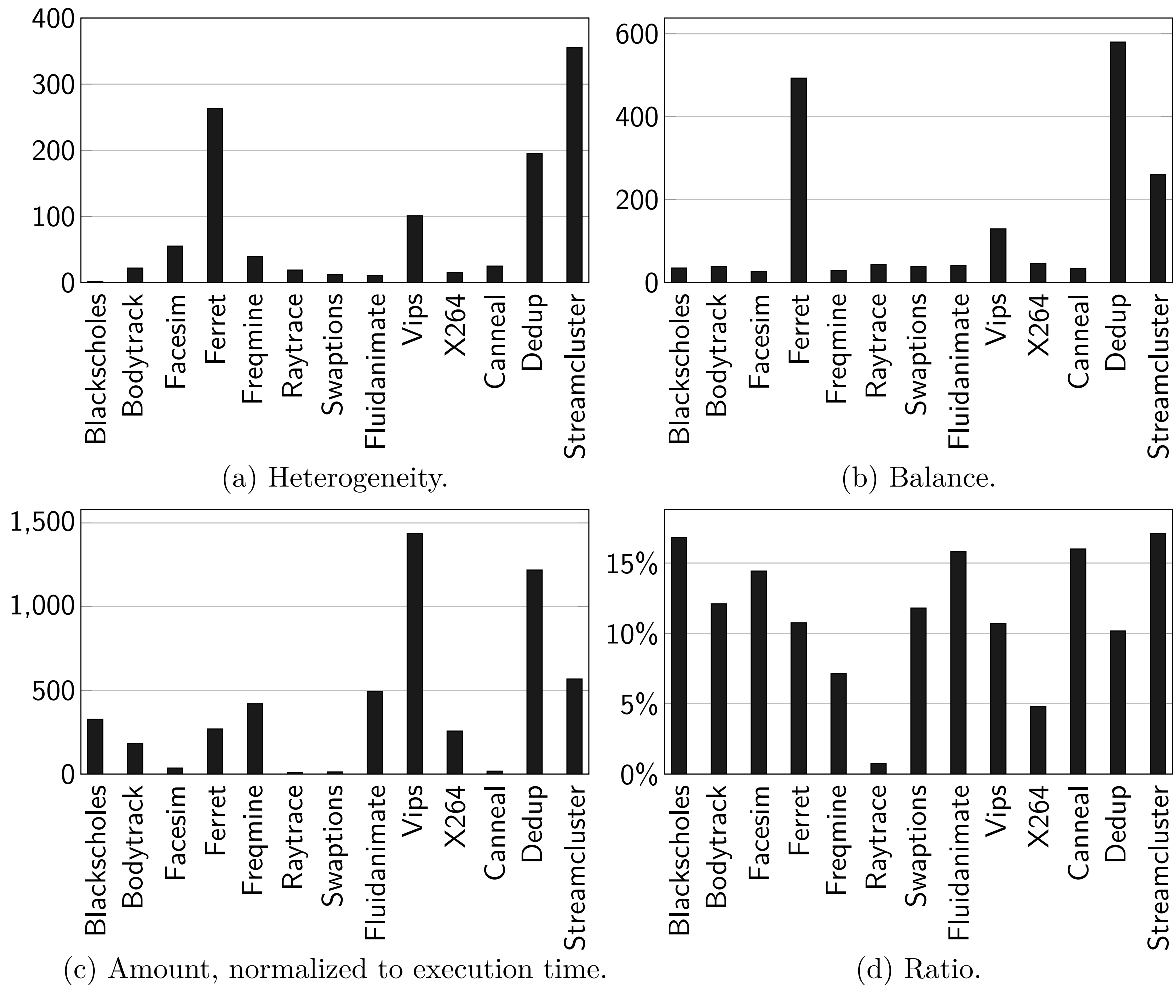
3.4.1.2 PARSEC Benchmarks

Figures 3.13 and 3.14 show the communication matrices and metrics of the PARSEC benchmarks, respectively. Only a minority of the PARSEC benchmarks have a high *heterogeneity*, indicating that PARSEC applications are generally less suitable for locality-based thread mapping. Benchmarks with a low heterogeneity mostly show initialization and all-to-all communication patterns. From the highly heterogeneous benchmarks, Fluidanimate shows communication between pairs of threads that are not neighbors, but farther apart, such as threads (0,8). Streamcluster and Ferret show a pipeline model, where groups of threads (pipeline stages) communicate among themselves.

Three PARSEC benchmarks, Ferret, Dedup, and Streamcluster, are significantly imbalanced due to their pipeline communication pattern. The load balance is again much lower than the communication balance (13.1 and 58.0 for Ferret, respectively). Similar to NAS-OMP, the communication *amount* differs widely between applications, but PARSEC benchmarks have a higher communication *ratio* in general.

3.4.1.3 NAS-MPI Benchmarks

Figures 3.15 and 3.16 show the communication matrices and metrics of the NAS-MPI benchmarks, respectively. The BT-MPI, LU-MPI, and SP-MPI benchmarks show similar communication patterns, with a large amount of communication between neighboring processes, such as processes (0,1), but also similar amounts of communication between



(a) Heterogeneity.

(b) Balance.

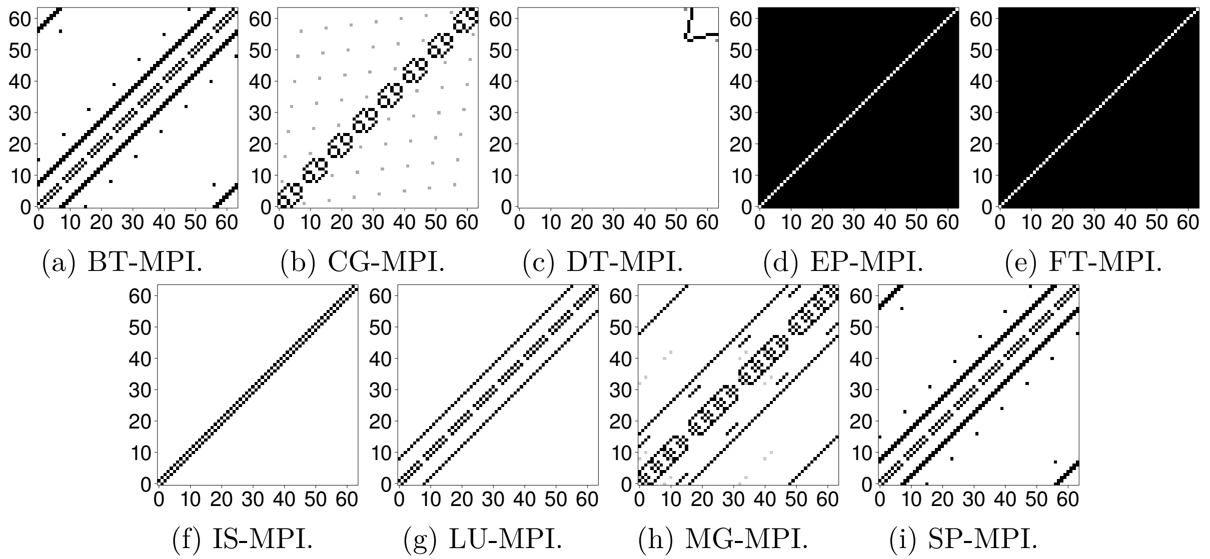
(c) Amount, normalized to execution time.

(d) Ratio.

Figure 3.14: Communication characteristics of the PARSEC benchmarks.

processes that are further apart, such as processes (0,4) and (0,20) in the case of BT-MPI and SP-MPI. EP-MPI and FT-MPI show very low amounts of communication between the processes, with maxima for neighboring processes. The pattern of IS-MPI shows similar amounts of communication for all pairs of processes, what we describe as an all-to-all pattern. In the DT-MPI benchmark, two pairs of processes, (16,19) and (16,31), perform the majority of communication. All other processes have a negligible amount of communication. In CG-MPI, clusters of 8 processes communicate with each other. For example, ranks 8–15 perform a large amount of communication among themselves. MG-MPI has a pattern similar to BT-MPI and SP-MPI, with a stronger focus on the neighboring processes.

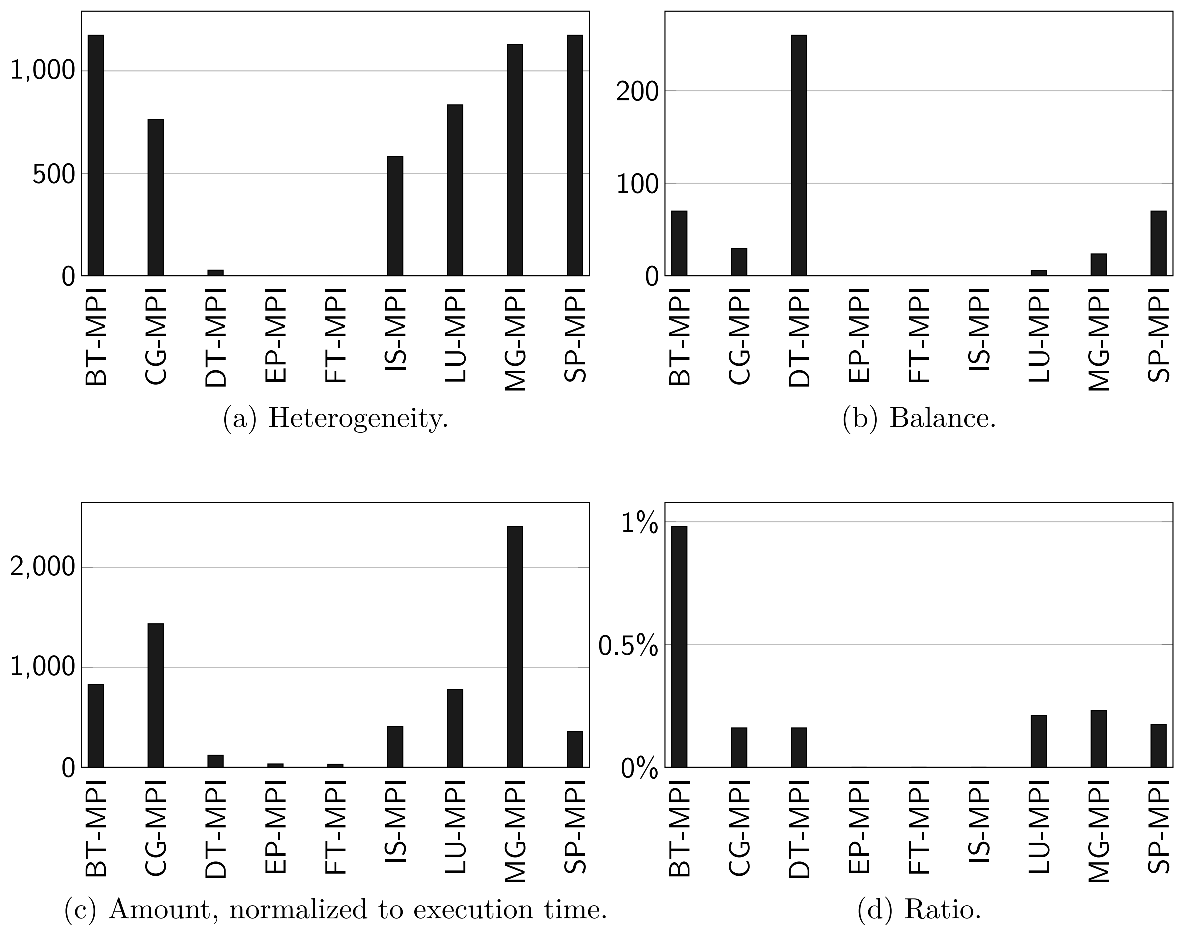
All NAS-MPI benchmarks except DT-MPI are well balanced. Although most benchmarks have large amounts of communication, the communication ratio shows that only a small percentage of memory accesses constitute communication, of less than 1% on average.



(a) BT-MPI. (b) CG-MPI. (c) DT-MPI. (d) EP-MPI. (e) FT-MPI.

(f) IS-MPI. (g) LU-MPI. (h) MG-MPI. (i) SP-MPI.

Figure 3.15: Communication matrices of the NAS-MPI benchmarks.



(a) Heterogeneity. (b) Balance.

(c) Amount, normalized to execution time. (d) Ratio.

Figure 3.16: Communication characteristics of the NAS-MPI benchmarks.

3.4.1.4 HPCC

The HPCC benchmark consists of 16 phases in total, labeled P1–P16 in Figures 3.17 and 3.18. Since it is a set of different HPC benchmarks, almost each phase has completely

different communication characteristics. The communication matrices of phases 1, 6, 7 and 15 are shown in Figure 3.17. Most phases show a complex communication behavior with several groups of threads communicating with each other. Additionally, on almost every of the 16 phases the communication behavior changes. Both the amount and ratio of communication are relatively low. The heterogeneity varies greatly between the phases. Despite the different heterogeneities, all phases are balanced, with similar amounts of communication for all processes.

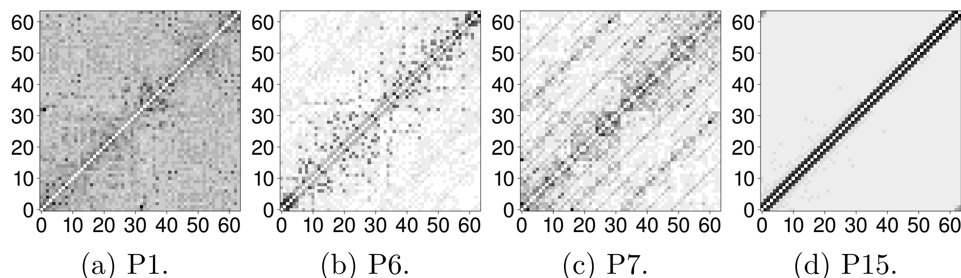
3.4.1.5 NAS-MZ Benchmarks

In the NAS Multi-Zone benchmarks, most communication is performed between the processes and between threads that are not direct neighbors, but farther apart. For all three Multi-Zone benchmarks, the communication pattern changes when modifying the number of processes and threads, even when maintaining the same total number of tasks. Similar to the NAS-OMP benchmarks, BT-MZ and SP-MZ are imbalanced since some of the threads do not communicate at all. Communication amount and ratio show that all 3 benchmarks are suitable for task mapping.

3.4.2 Dynamic Behavior

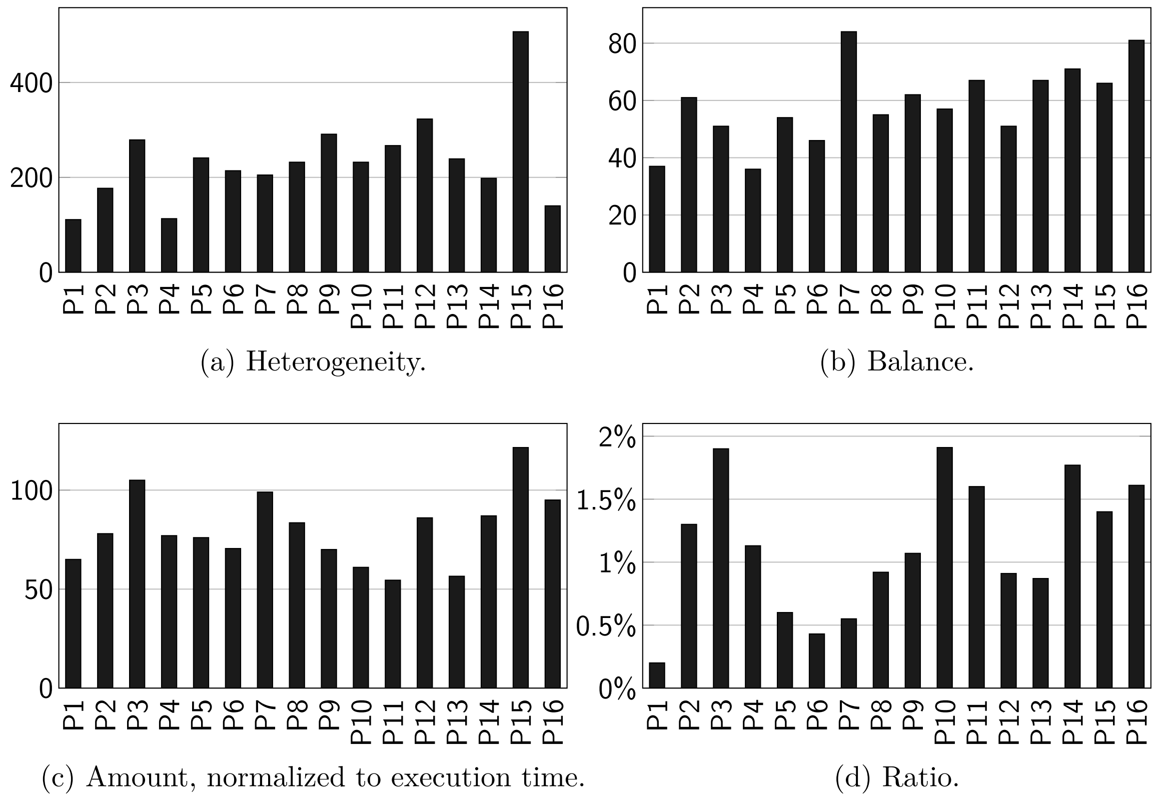
The dynamic communication behavior of the applications was evaluated with the dynamicity C_D introduced in Section 3.2.5. Figure 3.21 shows the number of communication phases per second of each benchmark, calculated by dividing the total number of detected communication phases by the execution time. Most benchmarks show a relatively stable communication behavior with less than 1 phase per second. Several of the PARSEC benchmarks show a very dynamic behavior, with more than 6 phases per second for Facesim and Ferret. We expect more improvements from task mapping with benchmarks that have a stable communication pattern with fewer phase changes, since fewer migrations are expected.

As an illustration of the dynamic behavior, consider Figure 3.22, which shows the dynamic behavior of the SP-OMP benchmark with the *D* input size. For a better visualization, we modified the application to execute only 2 time steps (out of 400), which corresponds to about 31000 time slices. Each black dot represents the amount of communication during a time slice, while the gray dots indicate the heterogeneity for a



(a) P1. (b) P6. (c) P7. (d) P15.

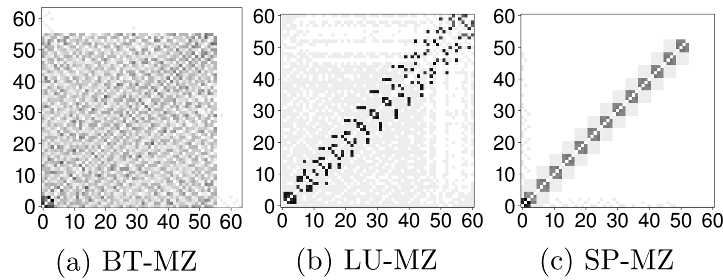
Figure 3.17: Communication matrices of various phases of the HPCC benchmark.



(a) Heterogeneity. (b) Balance.

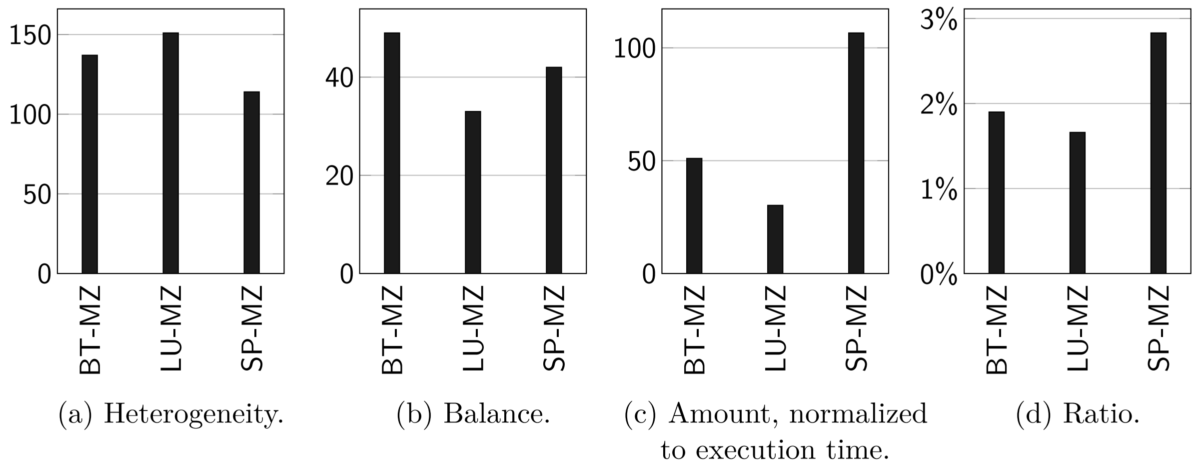
(c) Amount, normalized to execution time. (d) Ratio.

Figure 3.18: Communication characteristics of each phase of the HPCC benchmark.



(a) BT-MZ (b) LU-MZ (c) SP-MZ

Figure 3.19: Communication matrices of the NAS-MZ benchmarks.



(a) Heterogeneity. (b) Balance. (c) Amount, normalized to execution time. (d) Ratio.

Figure 3.20: Communication characteristics of the NAS-MZ benchmarks.

slice. We also show 3 communication matrices: 1 during the initialization and 2 matrices at the beginning and end of each time step, respectively. For a better visualization, the matrices are shown only with the first 8 threads.

We can differentiate several communication phases for SP-OMP. During the initialization of the application, a large amount of communication happens between all pairs of threads. Each time step spends its initial 80% of the execution time with a medium amount of communication and a high heterogeneity, corresponding to a communication pattern between neighboring threads. At the end of each time step, bursts of communication

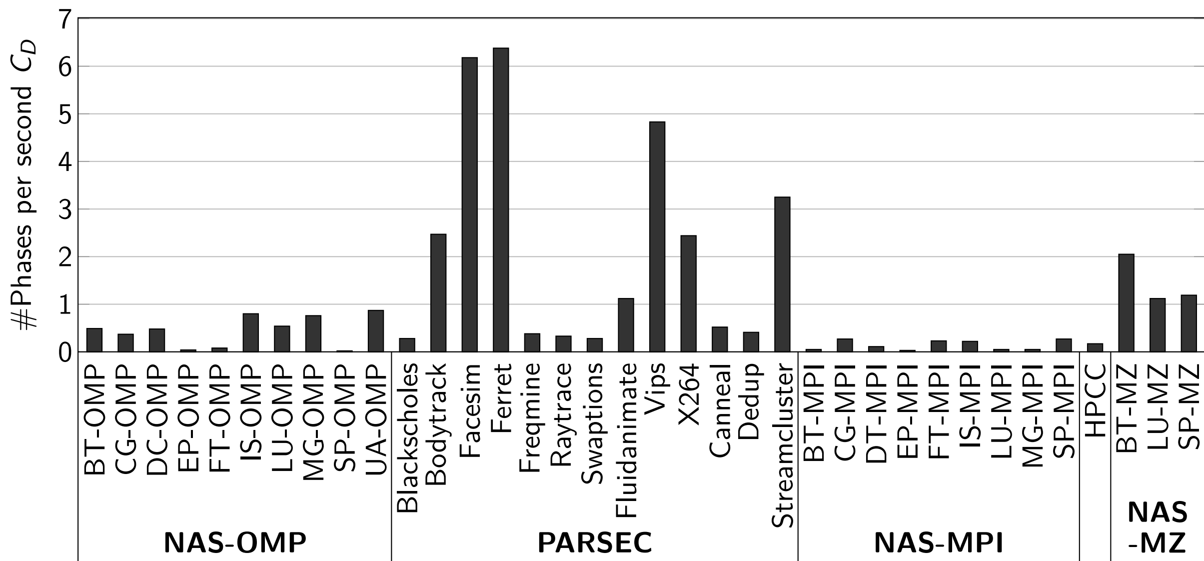


Figure 3.21: Number of communication phases per second execution time C_D.

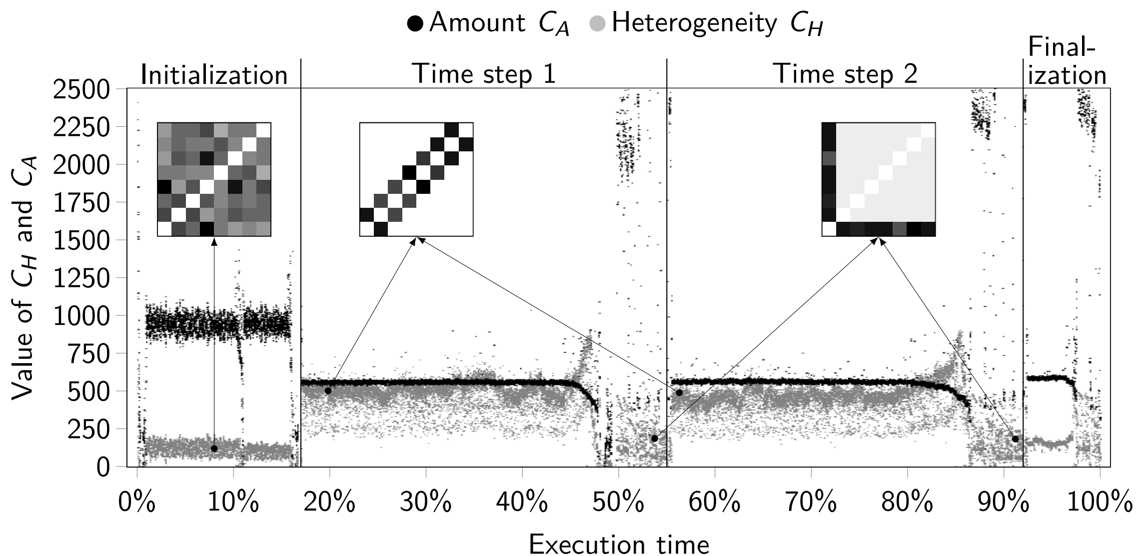


Figure 3.22: Dynamic communication behavior of the SP-OMP benchmark with the D input size for 2 time steps of the application. Each dot corresponds to the value of heterogeneity and amount of communication during a time slice. The arrows point to communication matrices (for the first 8 threads) at specific time slices.

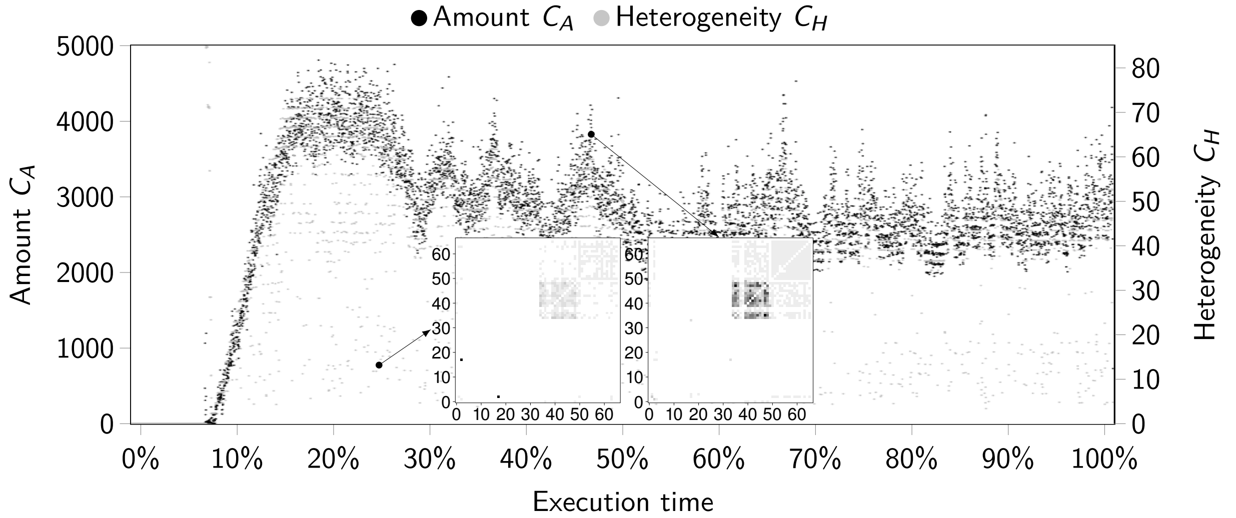


Figure 3.23: Dynamic communication behavior of the Ferret benchmark for 256 time steps of the application.

occur between the master thread and all other threads, corresponding to a reduction pattern. This regular communication behavior is then repeated for all the time steps of the application. During the finalization of the application, bursts of communication with a low heterogeneity occur.

In contrast to the very structured dynamic behavior of SP-OMP, the Ferret benchmark, shown in Figure 3.23, has a much more dynamic behavior. In the figure, we show the communication amount and heterogeneity for the first 256 time steps of the benchmark (out of 3500), resulting in about 7000 time slices. After the initialization up to 8% of the execution time, both metrics have a similar behavior, with large amounts of communication and a heterogeneity that varies quickly between 3 and 70. The reason for this behavior lies in the pipeline parallelization model of Ferret, which consists of 6 stages in total, where the first and last stage (consisting of only 1 thread each) handle input and output. As shown in Figure 3.13, only 1 of the 4 processing pipeline stages of Ferret actually communicates. This stage that communicates makes heavy use of memory accesses to a shared hash table (BIENIA et al., 2008), which results in the high amount of sharing within the group of threads. As the time steps of Ferret are very short (about 0.02 seconds per time step), the behavior changes very quickly.

Regarding the classification of dynamic memory access behavior discussed in Section 1.3.1, we can affirm that no application analyzed in this section changes its communication behavior between executions, as long as the input data and number of tasks remain the same. When modifying either input data or the number of tasks, the communication behavior changes substantially, as shown with the NAS-OMP benchmarks. Since the memory addresses themselves do not matter for communication detection, changing addresses do not constitute a dynamic behavior. On the other hand, many applications exhibit a varying degree of dynamicity during execution, as discussed in this section, mostly regarding algorithmic phase changes. These results indicate that communication traces

Table 3.2: Overview of benchmark suitability for task mapping considering 64 tasks. Darker cells indicate a higher value for the specified metric.

	NAS-OMP (B input size)										PARSEC													NAS-MPI										NAS-MZ		
Metric	BT-OMP	CG-OMP	DC-OMP	EP-OMP	FT-OMP	IS-OMP	LU-OMP	MG-OMP	SP-OMP	UA-OMP	Blackscholes	Bodytrack	Facesim	Ferret	Freqmine	Raytrace	Swaptions	Fluidanimate	Vips	X264	Canneal	Dedup	Streamcluster	BT-MPI	CG-MPI	DT-MPI	EP-MPI	FT-MPI	IS-MPI	LU-MPI	MG-MPI	SP-MPI	**HPCC**	BT-MZ	LU-MZ	SP-MZ
Hetero.																																				
Balance																																				
Amount																																				
Ratio																																				
Dynamic.																																				
Locality	✓	✓	✓			✓	✓	✓	✓	✓				✓								✓	✓	✓	✓				✓	✓	✓	✓	✓	✓	✓	✓
Balance	✓		✓				✓		✓					✓								✓	✓			✓										

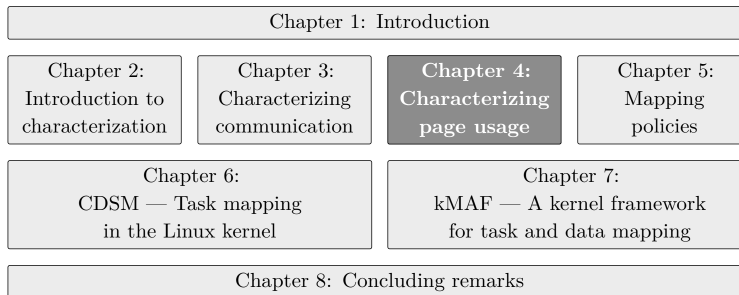
can be used for an improved mapping mechanism, and result in an accurate description of the behavior for multiple executions.

3.5 Summary of Communication Behavior

An overview of the communication metrics of all the benchmarks is shown in Table 3.2. In the first part of the table, darker colors indicate higher values of a specific communication metric, while the second part contains an estimation whether an application is suitable for locality-based or balanced-based task mapping. Summarizing the application behavior, we expect that applications with a high amount of communication, high communication ratio, and low dynamicity have a higher suitability for task mapping in general. Locality-based task mapping policies require applications with a high heterogeneity, while a high value of the balance metric (that is, when the application is imbalanced) favors balanced-based task mapping policies.

A majority of the benchmarks are suitable for locality-based mapping, while fewer benchmarks seem to be suitable for balance policies. Due to their less structured behavior, only a minority of PARSEC benchmarks appear to be suitable for task mapping. For the reasons mentioned before, even in these applications task mapping can improve the results of data mapping. Despite their relatively low communication ratios, most MPI-based applications show suitability for locality-based task mapping. In Section 5.2, we will evaluate the performance improvements of various task mapping policies and compare them to the characterization presented in this chapter.

4 CHARACTERIZING PAGE USAGE



This chapter introduces metrics and a methodology that analyzes the page usage of parallel applications to characterize their suitability for data mapping on NUMA architectures. In contrast to the communication behavior discussed in the previous chapter, where we discussed different ways to define and measure communication, page usage can be determined with much less ambiguity by analyzing memory accesses on the page size granularity. An additional difference is that the memory access behavior of an application in conjunction with the page allocation policy of the operating system always affects the data mapping. Therefore, the expected improvements of data mapping need to be evaluated by comparing the metrics to the values of the data mapping policy that the OS employs, such as a first-touch policy. In this chapter, we begin with a brief overview of qualitative page usage and then introduce the metrics. Finally, we analyze the parallel benchmarks in terms of these metrics.

4.1 Qualitative Page Usage

This section introduces and discusses the main concepts of the page usage of parallel applications qualitatively, focusing on how they interact with different data mapping policies.

4.1.1 Introduction to Page Usage

Since data mapping focuses on the placement and migration of memory pages, all discussion of page usage and its metrics uses the granularity of the page size of the architecture. Similar to the communication, page usage can be improved by considering two metrics, the *locality* and *balance* of memory accesses. For a locality-based policy, pages should be mapped to NUMA nodes from which they are most accessed. For a balance-based policy, pages should be mapped in such a way that all memory controllers handle a similar amount of memory accesses. Intuitively, pages that are accessed mostly from a single NUMA node are candidates for a locality-based policy, where these pages are mapped to the node with the most accesses to each page. For pages that are accessed in an equal way from all the nodes, a locality-based policy can usually not improve the overall memory access performance. However, these shared pages can be mapped in such a way that the number of memory accesses handled by different memory controllers is equalized with a balance-based policy, which can become an important factor in order not to overload some of the nodes.

Threads **T0** **T1** **T2** **T3**

Memory accesses

Page 1 **1** **25** **1** **1** Exclusive page

Page 2 **40** **35** **35** **20** Shared page

Figure 4.1: Example of memory access exclusivity for an application that accesses two pages and consists of four threads T_0–T_3.

4.1.2 Example of Page Usage

The two main concepts of page usage, locality and balance of memory accesses, are illustrated in this section. Both describe the access pattern to memory pages from threads and NUMA nodes.

4.1.2.1 Exclusivity and Locality

Exclusivity describes how many of the memory accesses to a page are caused by a single thread or a NUMA node. Pages with a high exclusivity, that is, pages that are accessed mostly from a single thread, are more suitable for locality-based policies since such a policy can reduce the memory access distance effectively. Figure 4.1 illustrates two pages with different exclusivities. Page 1 is mostly accessed by thread 1, with 25 of the 28 accessed performed by this thread. Therefore, this page has a high exclusivity and we call it an *exclusive page*. Page 2, on the other hand, has a much more distributed access pattern, with a similar number of accesses by all threads. This page has a low exclusivity and we refer to it as a *shared page*. Page 1 is more suitable for a locality-based mapping policy, but it has a lower overall impact on the memory accesses since it receives far fewer accesses than page 2. Since the locality of page 2 varies only slightly with the actual data mapping, it can be used to balance the memory accesses between memory controllers with a balance-based policy.

4.1.2.2 Balance

While exclusivity and locality can be discussed for a single page, the memory access balance needs to be evaluated by taking into account multiple pages. Two balance metrics can be analyzed: *page balance* and *memory access balance*. Figure 4.2 shows examples of both metrics for two different applications. Application 1 balances the number of pages fairly among the NUMA nodes, while application 2 allocates the majority of pages on node 1. This leads to a high imbalance for application 2. However, a high page balance does not necessarily imply a high memory access balance, because the number of accesses to each page might be different. An example of this situation is shown in the figure. Although application 1 has a high page balance, its memory accesses are imbalanced. Application 2 has the opposite behavior. Since a balance-based policy focuses on the memory accesses, simple mapping policies that are based on balancing pages (such as

interleave (MARCHETTI et al., 1995; KLEEN, 2004)) might not result in an optimal balance.

4.1.3 Common Types of Page Usage

This section discusses some common types of page usages and their impact on a first-touch mapping policy. We selected the matrix multiplication that was already presented in Section 3.1.6, since it contains examples of all types that we discuss. The source code[1] of the application is shown in Figure 4.3. The application has three main

[1] The full source code of the matrix multiplication application is also available at <https://github.com/matthiasdiener/pageusage-samples>

NUMA nodes N0 N1 N2 N3

Pages

Application 1 | 10 | 8 | 9 | 5 | Pages balanced

Application 2 | 1 | 10 | 1 | 1 | Pages imbalanced

Memory accesses

Application 1 | 5 | 50 | 5 | 5 | Memory accesses imbalanced

Application 2 | 100 | 90 | 90 | 50 | Memory accesses balanced

Figure 4.2: Example of memory access balance for two applications and four NUMA nodes N_0–N_3.

```
1  #define N 128
2  int A[N][N], B[N][N], C[N][N];
3
4  int main(int argc, char const *argv[])
5  {
6    int i=0, j=0, k=0;
7
8    memset(A, 1, N * N * sizeof(int)); //initialize matrices
9    memset(B, 2, N * N * sizeof(int));
10   memset(C, 0, N * N * sizeof(int));
11
12   #pragma omp parallel for
13   for(i=0; i<N; i++)
14     for(j=0; j<N; j++)
15       for(k=0; k<N; k++)
16         C[i][j] += A[i][k] * B[k][j];
17   return 0;
18 }
```

Figure 4.3: Matrix multiplication code.

structures, `A`, `B`, and `C`, representing square matrices of size $N \times N$. The matrices are initialized with the `memset()` function (Lines 8–10). After initialization, the application calculates $\texttt{C} = \texttt{A} \times \texttt{B}$ in parallel (Lines 12–16).

We generated a memory access trace of the application when executing it with 4 threads, recording every memory access of every thread for each page that the application accesses. Memory accesses are recorded with a custom Pin tool. Table 4.1 shows the memory access profile for a subset of pages that the application uses[2]. For each page, we show the address of the page, the structure it is associated to (one of the matrices, `A`, `B`, or `C`, or a thread's stack), which thread performed the first access to the page, and the number of memory accesses from each thread to the page. This page usage profile exposes several issues for data mapping, which will be discussed in this section. Both locality-based and balance-based policies are affected. For the discussion, consider that the application is executing on a system with 4 NUMA nodes, and that each thread is running on a different node.

4.1.3.1 Initialization from a Single Thread

Analyzing the first access column shows that all pages were first accessed from thread 0. This means that with the default first-touch mapping policy, all pages will be allocated on the NUMA node where thread 0 was executing when the application was starting up, before performing the actual work. All 162 pages were first accessed by thread 0. This contradicts the ideas behind the introduction of the first-touch mapping policy, which was developed as an improvement of memory access locality compared to a round-robin mapping of pages to NUMA nodes (MARCHETTI et al., 1995). Due to the way Linux allocates static data, removing the `memset()` calls results in the same first access behavior.

This behavior is an opportunity to improve both the locality and balance of the memory accesses. For example, page 1558 is mostly accessed by thread 1, but it is allocated on the NUMA node of thread 0. Migrating this page to the NUMA node of thread 1 increases the locality of accesses to the page. Furthermore, the NUMA node of thread 0 will handle all main memory accesses of the application, creating an imbalance on its memory controller. By distributing the pages more fairly among the NUMA nodes, a balance-based policy can improve the overall load on the memory controllers.

4.1.3.2 Stacks of Multithreaded Applications

A related problem is shown in the allocation of the stacks of each thread. In Linux, multithreaded applications share the same virtual address space, but also contain data that is private to each thread. To create a multithreaded application on Linux, the runtime system uses the `clone()` or `fork()` system calls, which are executed in the context of the calling thread. This calling thread is usually the master thread of the application, that is, thread 0. The private data of the newly created thread needs to be zeroed (such as the stack, to prevent information leakage) or initialized (such as the Thread-Local Storage (TLS) (DREPPER; MOLNAR, 2002)). For this reason, the created thread's

[2] 20 out of 162 pages are shown.

Table 4.1: Page usage profile of the matrix multiplication application, executing with 4 threads, T_0–T_3. The table shows the address, structure name, the thread that performed the first access to the page, and the number of accesses to the page for each thread for selected pages.

			Number of memory accesses			
Address	Structure	First access	T_0	T_1	T_2	T_3
1537	B	T_0	17,222	9,347	8,173	9,054
1538	B	T_0	17,501	9,667	7,626	9,149
1553	C	T_0	95,869	1,412	1,146	1,382
1557	C	T_0	12,544	56,598	0	0
1558	C	T_0	256	81,028	0	0
1561	C	T_0	256	5,254	77,738	0
1564	C	T_0	256	0	70,835	0
1565	C	T_0	256	0	4,454	76,489
1566	C	T_0	256	0	0	57,033
1572	A	T_0	131,328	0	0	0
1573	A	T_0	6,400	28,445	0	0
1575	A	T_0	256	42,619	0	0
1577	A	T_0	256	2,594	38,912	0
1578	A	T_0	256	0	26,010	0
1581	A	T_0	256	0	2,195	38,343
1582	A	T_0	256	0	0	28,546
34359738367	Stack T_0	T_0	5,381,218	1,230,331	1,017,819	1,177,382
34359703002	Stack T_1	T_0	2	1,688,110	0	0
34359700953	Stack T_2	T_0	2	0	1,395,867	0
34359698904	Stack T_3	T_0	2	0	0	1,614,711
Total	—	—	6,517,798	3,527,898	2,917,345	3,427,447

private data is first accessed by the master thread, leading to an unfavorable page mapping with a first-touch policy.

The resulting data mapping is inefficient both in terms of locality and balance. It overloads the NUMA node that runs the master thread with the stack accesses of all threads, leading to an imbalance. As shown in the table, the vast majority of memory accesses to each stack are performed by the thread that owns the stack, leading to a bad locality of these accesses. These issues are further aggravated by the fact that the stack receives a lot of memory accesses compared to the pages of the other structures. An improved data mapping policy can help to improve the balance and locality of the stack usage.

4.1.3.3 Exclusive Pages

As mentioned before, the memory pages used for the stacks have a high exclusivity since they are accessed mostly from a single thread. Two of the matrices of the matrix

multiplication are also exclusive, the `A` and `C` structures. For example, page 1553 is accessed mostly by thread 0, while page 1558 is accessed mostly by thread 1. This pattern is repeated for the whole structure. Some of the pages that are on the border between two areas of the matrix have an overlap of accesses by two different threads, such as page 1557, which is accessed by threads 0 and 1. Despite this overlap, the page is still relatively exclusive, since more than 80% of accesses are performed by thread 1. For exclusive pages, the most appropriate data mapping policy is based on increasing locality. By placing pages on NUMA nodes that perform the most accesses to them, the overall memory access latency can be reduced.

This example also illustrates another important aspect. Performing a locality-based mapping policy can also increase the balance of memory accesses in some cases. Considering the `C` matrix as an example, which consists of 16 pages in total, 7 of which are shown in Table 4.1. In a system with 4 NUMA nodes, a locality-based policy would map 4 consecutive pages of `C` to the NUMA node of each thread, increasing the locality of the accesses. In addition, this mapping also maximizes the balance of the mapping compared to the first-touch policy, which would map all 16 pages to the NUMA node of thread 0. The `A` matrix has very similar behavior. This shows that for these two structures, a locality-based policy also increases the balance.

4.1.3.4 Shared Pages

The `B` matrix has a different structure of memory accesses. All 16 pages of this structure have an access pattern similar to the 2 pages shown in Table 4.1. These pages are accessed by all threads in a similar manner, with only small differences between the minimum and maximum numbers of accesses. This indicates that these pages are shared between threads, and a locality-based policy can improve the overall memory access locality only slightly. However, these pages are candidates for a balance-based policy and can be used to distribute the load on the memory controllers more equally. For example, `B` could be distributed equally among all the NUMA nodes, or its pages could be migrated from an overloaded node to reduce its load.

4.1.3.5 Overall Balance

Finally, it is also important to consider the overall memory access balance of the whole application. The number of memory access of each thread is shown in the *Total* row of Table 4.1. The numbers show that thread 0 is performing the majority of memory accesses of all threads (about 40%). By placing the pages that the threads access in a more balanced way, it is possible to improve the overall load on the memory controllers and achieve a fairer balance.

4.1.3.6 Summary

We conclude that the default first-touch policy of modern operating systems presents inefficiencies, both for the locality and balance of memory accesses, and optimizes for neither metric. This indicates that large improvements can be achieved by improved data

mapping policies. Section 4.2 will formalize the intuitive ideas of exclusivity, locality and balance that were introduced here.

4.1.4 Data Mapping for MPI Applications?

In contrast to multithreaded applications that are based on shared memory programming models such as OpenMP and Pthreads, parallel applications based on message passing models such as MPI are less interesting for data mapping. The MPI runtime is based on multiple processes that do not share the virtual address space and whose data is mostly private to each process (FRIEDLEY et al., 2013). In such a model, the first-touch policy already results in a perfect locality of memory accesses, unless processes are migrated between NUMA nodes. The only data shared between processes is the small memory segment used for communication, which is also allocated on local nodes in most cases (EKMAN et al., 2005), with further optimizations having only a negligible impact on performance (EKMAN et al., 2005).

For this reason, we will only consider parallel applications based on shared memory models for data mapping. More specifically, we consider the NAS-OMP and PARSEC benchmark suites and the Ondes3D and HashSieve applications. For MPI applications running on NUMA architectures, task mapping is still relevant and needs to avoid migrations between NUMA nodes (EKMAN et al., 2005) in addition to optimizing communication. This will be discussed in more detail in Section 5.

4.2 Quantitative Page Usage

In this section, we introduce metrics to describe the memory access behavior of parallel applications on NUMA systems. We present metrics for the suitability of applications for locality-based and balance-based policies. We also discuss dynamic application behavior through the changes to the exclusivity during execution. Finally, we introduce metrics for the minimum memory usage of an application, below which no gains from data mapping can be expected, and a metric to describe the locality of a given data mapping. We describe the memory access behavior to pages with the spatial, volume and temporal components, similar to the communication behavior.

4.2.1 Memory Access Exclusivity

The potential for locality-based data mapping of a page is proportional to the amount of memory accesses from a single NUMA node. That is, if a page is mostly accessed from the same node, it has more potential for locality-based mapping than a page that is accessed from several nodes. We call the highest number of memory accesses to a page from a single node compared to the number of accesses from all nodes the *page exclusivity* E_{Page}. The higher the exclusivity of a page, the higher its potential for locality-based mapping. E_{Page} is calculated with Equation 4.1, where $MemAcc[p][n]$ is the number of

memory accesses to page p from node n, and N is the number of NUMA nodes. The max function returns the highest number of memory accesses to a page from a node.

$$E_{Page}[p] = \frac{\max(MemAcc[p])}{\sum_{n=1}^{N} MemAcc[p][n]} \tag{4.1}$$

The exclusivity is minimal when a page has exactly the same number of accesses from all nodes. In this case, the exclusivity is given by $1/N$. The exclusivity achieves its maximum value of 1 when all accesses to the corresponding page originate from the same NUMA node.

Besides the exclusivity of a page, the number of memory accesses to it has to be considered as well. The higher the number of accesses to a page, the higher its potential for data mapping. We take the number of memory accesses into account by measuring the exclusivity for the whole application. To calculate this *application exclusivity* E_{App}, we scale the exclusivity of each page with the number of memory accesses to it, and divide this value by the total number of memory accesses. This operation is shown in Equation 4.2, where P is the total number of pages. Like the page exclusivity, the minimum and maximum of the application exclusivity is $1/N$ and 1, respectively, when all pages are fully shared or fully exclusive.

$$E_{App} = \frac{\sum_{p=1}^{P} (E_{Page}[p] \times \sum_{n=1}^{N} MemAcc[p][n])}{\sum_{p=1}^{P} \sum_{n=1}^{N} MemAcc[p][n]} \tag{4.2}$$

4.2.2 Memory Access Balance

Our second metric, *memory balance*, is used to analyze the suitability of a parallel application for balance-based data mapping. Memory balance is important in applications where the memory accesses are performed in such a way that some memory controllers are overloaded, while others are idle. To measure the *page balance* B_{Pages}, we introduce a new vector, $NodePages$, which consists of N elements. Each element $NodePages[n]$ contains the number of pages mapped to node n. Equation 4.3 calculates the page balance for all the pages that an application uses. The equation has a similar structure as the communication balance presented in Section 3.2.2.

$$B_{Pages} = \left(\frac{\max(NodePages)}{\sum_{n=1}^{N} NodePages[n]/N} - 1 \right) \times 100\% \tag{4.3}$$

The balance is maximum (with a value of 0%), when all nodes store the same number of pages. Higher values indicate a higher page imbalance. The highest imbalance is reached (with a value of $(N-1) \times 100\%$) when one node stores all the pages.

Since not all pages have the same number of memory accesses, maximizing load balance considering Equation 4.3 may not improve the balance of memory controllers. To achieve an improved balance, we need to consider the number of memory accesses to each

page. We store the number of memory accesses to each NUMA node in the $NodeAcc$ vector, as shown in Equation 4.4.

$$NodeAcc[n] = \sum_{p=1}^{P} MemAcc[p][n] \tag{4.4}$$

We can then calculate the *memory access balance* B_{Acc} of the application with Equation 4.5.

$$B_{Acc} = \left(\frac{\max(NodeAcc)}{\sum_{n=1}^{N} NodeAcc[n]/N} - 1 \right) \times 100\% \tag{4.5}$$

If the memory accesses are equally distributed among the nodes, B_{Acc} has a value of 0%. If all memory accesses are satisfied from a single node, the imbalance is maximized, and B_{Acc} has a value of $(N-1) \times 100\%$.

4.2.3 Total Memory Usage

Another requirement for data mapping is that the memory usage of the application must be significantly higher than the cache size of the system. If the memory that an application uses fits into the processor caches, fewer improvements from a data mapping policy can be expected, as most memory accesses can be filtered by the caches. We use Equation 4.6 as a lower bound, where $PageSize$ is the size of each page, C is the total number of caches in the system and $CacheSize[i]$ is the size of Cache i.

$$\begin{aligned} Mem_{Usage} &= P \times PageSize \\ Mem_{Usage} &> \sum_{i=1}^{C} CacheSize[i] \end{aligned} \tag{4.6}$$

In our experiments, this affects two of the applications we discuss in this thesis, EP-OMP and Swaptions, whose memory usage is 66 MByte and 7 MByte, respectively, for 64 threads and the input sizes we use. This small memory usage fits into the caches of many current systems. All other applications have a memory usage of at least several hundred MByte and are therefore not affected by this lower bound.

4.2.4 Dynamic Behavior

Similar to the dynamic communication behavior presented in Section 3.2.5, we divide the execution of each application into time slices of a fixed size to analyze the dynamic page usage of the parallel applications. For each time slice, we calculate for each page p if it needs to be migrated to the NUMA node with the most accesses. This behavior is expressed in Equation 4.7.

$$Migrate[p] = \arg\max(MemAcc_{cur}[p]) \neq \arg\max(MemAcc_{cur-1}[p]) \tag{4.7}$$

As in the temporal communication detection, we use time slices of 10 ms. This value is used to characterize the dynamic access behavior to pages and does not necessarily indicate that a page needs to be migrated. As another indicator of dynamic behavior, we also calculate the application exclusivity for each time slice. By determining the number of migrations with Equation 4.7, we can then calculate the *Dynamicity* of the page usage $Page_{Dyn}$ by normalizing it to the total execution time of the application, as shown in Equation 4.8. We expect applications with a higher dynamicity that require more migrations during execution to be less suitable for data mapping due to the overhead of the page migrations.

$$Page_{Dyn} = \frac{\#migrations}{execution\ time} \tag{4.8}$$

4.2.5 Locality of a Page Mapping

To compare different mapping policies to each other in terms of their locality, we first use Equation 4.9 to define if a single page is located on the node with the most accesses to it[3]. In the equation, $CurNode[p]$ is the NUMA node where page p is currently located, and the $\arg\max$ function returns the NUMA node with the highest number of accesses to p. Here, we assign 1 to the locality value if the page is located on the NUMA node with the highest number of accesses to the page, and 0 otherwise.

$$Loc_{Page}[p] = \begin{cases} 1, & \text{if } \arg\max(MemAcc[p]) = CurNode[p] \\ 0, & \text{otherwise} \end{cases} \tag{4.9}$$

We scale the locality of each page with the number of accesses to it to calculate the locality of a page mapping for the entire application, expressed with Loc_{App}. Equation 4.10 shows this operation, where P is the number of pages. The minimum for Loc_{App} is 0%, when no page is mapped to the node with the highest accesses to it, and 100% when all pages are mapped to the node with the most accesses.

$$Loc_{App} = \frac{\sum_{p=1}^{P} \left(Loc_{Page}[p] \times \sum_{n=1}^{N} MemAcc[p][n] \right)}{\sum_{p=1}^{P} \sum_{n=1}^{N} MemAcc[p][n]} \tag{4.10}$$

4.2.6 Summary of Page Usage Metrics

Table 4.2 contains a summary of the page usage metrics that were introduced in this section, showing which component of the behavior they describe, which type of data mapping they are more suitable for, and if a high or low value of the metric indicates higher suitability for the task mapping.

[3] The balance of mappings can be compared directly with B_{Acc}.

Table 4.2: Overview of the page usage metrics introduced in this section.

Name	Symbol	Granularity	Describes	Mapping policy	Suitability
Page exclusivity	E_{Page}	Page	Structure	Locality-based	High
App. exclusivity	E_{App}	Application	Structure	Locality-based	High
Page balance	B_{Pages}	Application	Structure	Balance-based	High
App. balance	B_{Acc}	Application	Structure	Balance-based	High
Total mem. usage	Mem_{Usage}	Application	Volume	Both	High
Dynamic page usage	$Page_{Dyn}$	Application	Temporal	Both	Low
Locality	Loc_{App}	Application	Comparison	Locality-based	High

4.3 Data Mapping Policies

To evaluate the impact of data mapping on the performance of the parallel applications, we compare several mapping policies to the default first-touch policy. These policies will be presented in this section.

4.3.1 The Policies

The following mapping policies were evaluated: *Random*, *RoundRobin*, *Interleave*, *Locality*, *Remote*, *Balanced*, and *Mixed*. The Interleave policy is available in many operating systems, such as via the `numactl` tool in Linux (KLEEN, 2004). Locality is a policy similar to previous mechanisms that focus on locality improvements in NUMA systems (MARATHE; THAKKAR; MUELLER, 2010; DIENER et al., 2014). Remote is the opposite of Locality. Balanced and Mixed are two new mapping approaches. Balanced distributes pages in such a way that all memory controllers resolve the same number of memory accesses. We also introduce a mixed policy, which presents a trade-off between locality and balance. In the following description of the mapping policies, $node[p]$ represents the NUMA node of a page p, and N represents the total number of nodes. Consider that $p.AccNode$ is a vector, where each element $p.AccNode[n]$ contains the number of memory access to page p from node n.

4.3.1.1 Random

In the *Random* mapping, each page gets assigned randomly to a NUMA node. We use this mapping to validate the importance of data mapping and because it is the mapping that is most independent from the memory access behavior of the application. Equation 4.11 calculates this policy, where the random() function returns a random integer.

$$node[p] = \text{random}() \bmod N \tag{4.11}$$

4.3.1.2 RoundRobin

In the *RoundRobin* mapping, pages get allocated to NUMA nodes in the order that they are first accessed. The node for a page p gets assigned with the following equations:

$$\begin{aligned} node[p] &= count \bmod N \\ count &= count + 1 \end{aligned} \tag{4.12}$$

This mapping ensures that the same number of pages gets assigned to each node, independent of the memory access behavior and the addresses that the pages are allocated at.

4.3.1.3 Interleave

In the traditional *Interleave* policy, pages get assigned to NUMA nodes according to their address. Usually, the lowest bits of the page address are used to determine the node. Equation 4.13 describes this behavior, where $\mathrm{addr}(p)$ returns the page address of page p.

$$node[p] = \mathrm{addr}(p) \bmod N \tag{4.13}$$

This policy ensures that memory accesses to consecutive addresses are distributed among the nodes. Similar to the RoundRobin policy, it also maps equal numbers of pages to each NUMA node in case pages have contiguous addresses.

4.3.1.4 Locality

In the *Locality* policy, each page gets assigned to the NUMA node with the most memory accesses to the page. This policy ensures that the highest possible number of memory accesses are resolved by the local NUMA node. However, it does not take the balance between nodes into account and can lead to overloaded nodes. Equation 4.14 describes this behavior, where $\arg\max(p)$ returns the NUMA node with the most accesses to page p.

$$node[p] = \arg\max(p.AccNode) \tag{4.14}$$

4.3.1.5 Remote

In the *Remote* policy, each page gets placed on the NUMA node that has the fewest accesses to that page. This mapping represents a worst-case for data mapping and is used to further evaluate its importance on parallel applications. Equation 4.15 describes this behavior, where $\arg\min(p)$ returns the node with the fewest accesses to page p.

$$node[p] = \arg\min(p.AccNode) \tag{4.15}$$

Algorithm 4.1: Calculate the *Balanced* mapping.

```
Input: p.AccNode[n]: number of accesses to page p from node n; N: number of
       NUMA nodes
Output: node[p]: mapping of pages to nodes
1  sort pages by number of accesses;
2  TotalAcc = 0;
3  for each Page p do
4  |  p.TotalAcc = 0;
5  |  for each Node n do
   |  |  // total memory accesses per page
6  |  |  p.TotalAcc += p.AccNode[n];
7  |  end
   |  // total memory accesses
8  |  TotalAcc += p.TotalAcc;
9  end
10 for each Node n do
11 |  AccNode[n] = 0;
12 end
13 for each Page p do
   |  // find node with the most accesses to p that is not overloaded
14 |  n = node with the most accesses to p;
15 |  while AccNode[n]/TotalAcc > 1/N do
16 |  |  n = node with the next highest accesses to p;
17 |  end
   |  // map page p to node n and update values
18 |  node[p] = n;
19 |  AccNode[n] += p.TotalAcc;
20 end
```

4.3.1.6 Balanced

In the *Balanced* mapping, we maximize the balance between the nodes, while still taking into account the locality of each page. Algorithm 4.1 calculates this mapping. In the algorithm, we first sort the list of pages by the number of accesses. Then, we map each page to the node with the most accesses to the page that is not overloaded.

Balanced and Locality are opposite policies: Balanced optimizes memory access balance over locality, while the Locality policy maximizes local accesses at the expense of memory balance. For this reason, these policies can determine which of the metrics is more important for the performance of parallel applications.

4.3.1.7 Mixed

In the *Mixed* mapping, we combine the locality and balance metrics. For each page, if its exclusivity E_{Page} is above a threshold $minExcl$, we allocate the page on the node with the highest number of accesses, as in the Locality policy. If the exclusivity is below

Table 4.3: Input for the data mapping example for an application consisting of four threads. For each page, the table shows which thread accessed a page first as well as the number of memory accesses to that page by each of the four threads.

		Number of memory accesses			
Page	First access	T_0	T_1	T_2	T_3
0	T_0	1	0	1,000	0
1	T_0	1	1,000	0	0
2	T_0	1,000	0	0	0
3	T_0	1,000	0	0	50
Total	—	2,002	1,000	1,000	50

the threshold, we allocate the page on the node given by the Interleave mapping. This behavior is formalized in Equation 4.16.

$$node[p] = \begin{cases} \arg\max(p.AccNode), & \text{if } E_{Page}[p] > minExcl \\ \text{addr}(p) \bmod N, & \text{otherwise} \end{cases} \tag{4.16}$$

This mapping provides a trade-off between locality and balance. For $minExcl =$ 100%, Mixed is equivalent to Locality, for $minExcl = 0\%$, it is identical to the Interleave policy.

4.3.1.8 First-touch

The previously presented mapping policies will be compared to a *First-touch* policy, where each page gets allocated on the node that accessed the page for the first time (MARCHETTI et al., 1995). Pages are not migrated during execution. This policy is the default for most operating systems, such as Linux (LANKES; BIERBAUM; BEMMERL, 2010), Solaris (ORACLE, 2010), and Windows (PAAS, 2009), among others.

4.3.2 Example of Policy Behavior

As an example of the behavior of the data mapping policies, consider the page usage example presented in Table 4.3. In this example, a parallel application with 4 threads T_0–T_3accesses 4 pages. All pages are accessed first by thread T_0, and the table shows the total number of memory access by each thread. Further assume that we want to execute this application on a system with 4 NUMA nodes, where each thread is mapped to a different NUMA node with a Compact task mapping (that is, thread 0 executes on node 0, thread 1 on node 1, etc.).

Figure 4.4 shows the behavior of the different data mapping policies for this example. In the figure, gray squares represent the NUMA nodes, while the black circles represent the pages. The First-touch policy allocates all pages on the first NUMA node, since all pages get accessed first by thread 0, resulting in low memory access locality and a high

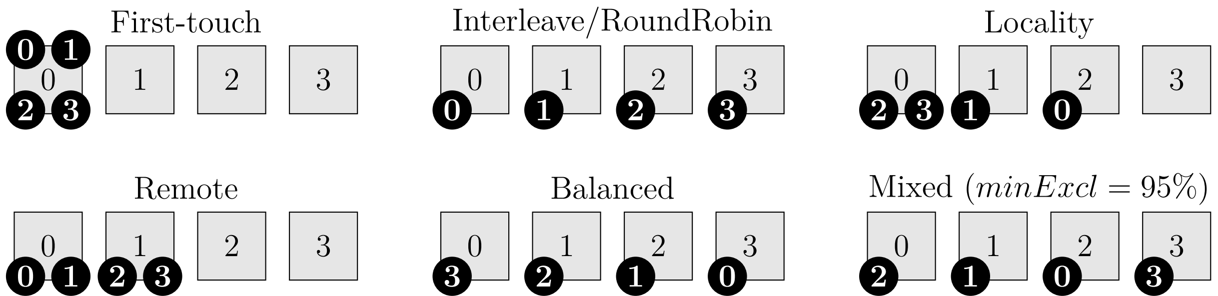


Figure 4.4: Example of data mapping policy behavior. Gray squares represent NUMA nodes, black circles represent pages.

imbalance. The Interleave and RoundRobin policies result in the same mapping, which has a good balance but a low locality. The Locality policy results in a very high locality, as expected, but due to the imbalanced application behavior does not distribute pages fairly among the nodes (node 3 stores no page, for example). The Remote policy results in a mapping with a low locality and high imbalance. The Balanced mapping distributes pages fairly among nodes, but pages 0–2 are mapped to nodes which have no accesses to them. The Mixed policy (shown for a locality threshold *minExcl* of 95%), results in a slightly lower locality compared to the Locality mapping, but results in a very high page and memory access balance.

4.4 Page Usage of the Benchmarks

For the evaluation of the page usage of the NAS-OMP and PARSEC benchmarks in this section, we execute each application with 64 threads and record each memory access of each thread on the page granularity with the numalize memory access tracer presented in Section 2.3. We use a default page size of 4 KByte. We start with an analysis of the global application behavior, followed by a discussion of the dynamic page usage.

4.4.1 Global Behavior

We begin with a discussion of the global application behavior, that is, the behavior during the whole execution of the application.

4.4.1.1 Exclusivity

The application exclusivity E_{App} is presented in Figures 4.5 and 4.6 for two configurations: (i) our baseline, with 64 threads on 4 NUMA nodes (threads are assigned to NUMA nodes such that the exclusivity is highest), (ii) 64 NUMA nodes (1 thread per node), to show the inherent exclusivity of the applications. Results are further divided into three page sizes, 4 KByte (which is the default page size in 32-bit and 64-bit x86 architectures (INTEL, 2013b)), 2 MByte (which is the highest page size supported by 32-bit

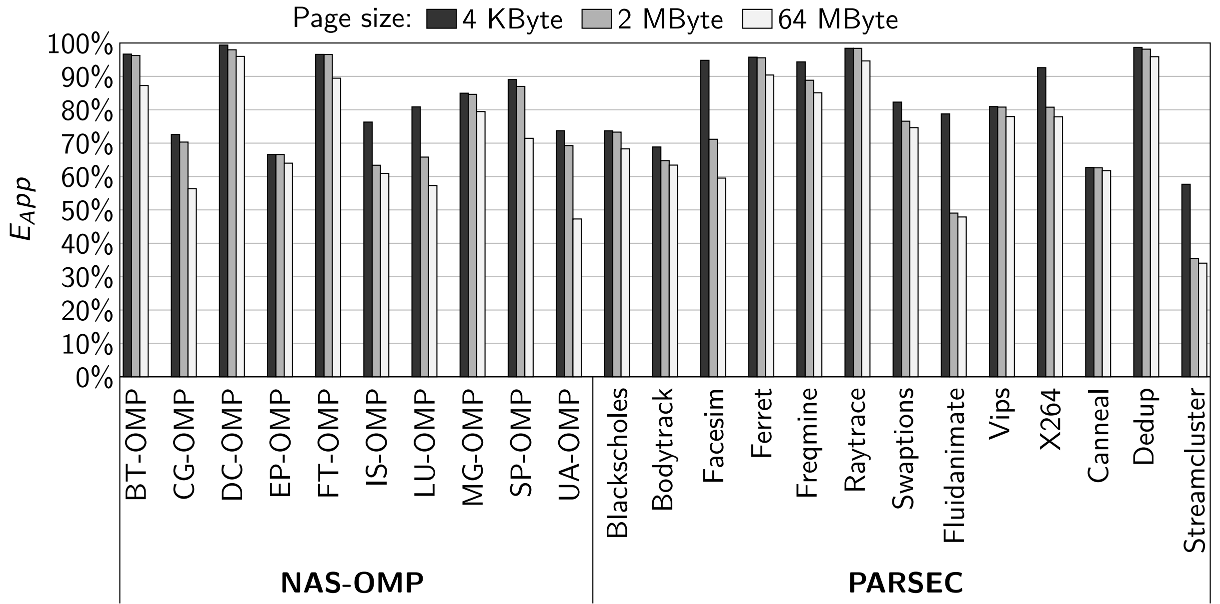


Figure 4.5: Application exclusivity E_{App} for 4 NUMA nodes and different page sizes.

x86 with the Physical Address Extension (PAE) mode (INTEL, 2013b), and 64 MByte (as an example of how the exclusivity changes when increasing page sizes further).

Most applications have a high exclusivity, even for 64 NUMA nodes, which demonstrates the importance of a locality-based mapping policy. Several benchmarks, such as EP-OMP, Canneal, and Streamcluster, have a lower exclusivity however. Even when increasing the page size or the number of NUMA nodes, the exclusivity only decreases slightly for most benchmarks. A slight decrease of exclusivity is expected for larger pages, as more shared data is stored in the same pages.

When increasing the number of NUMA nodes, more shared pages will be located on different nodes, lowering the exclusivity. On average, the application exclusivity for the 4 KByte, 2 MByte and 64 MByte page sizes is 83.3%, 77.1% and 71.3% (for 4 nodes), and 74.0%, 64.4% and 29.0% (for 64 nodes), respectively. Although the reduction is low in most cases, these results indicate that with larger pages or more NUMA nodes, fewer improvements from a locality-based data mapping can be expected. Nevertheless, in the most relevant case for data mapping, where the number of threads is much higher than the number of NUMA nodes, we expect still high improvements even with pages that are much larger than the current default of a few KByte.

4.4.1.2 Balance

Figure 4.7 visualizes the page and memory access balance of the first-touch policy and a system with 4 NUMA nodes. For all NAS-OMP benchmarks except EP-OMP, as well as some of the PARSEC benchmarks, the policy distributes the pages fairly between the nodes. On the other hand, comparing the results for the page and memory access balance shows that a high page balance does not necessarily result in a high memory access balance. For example, UA-OMP with first-touch has a very good page balance

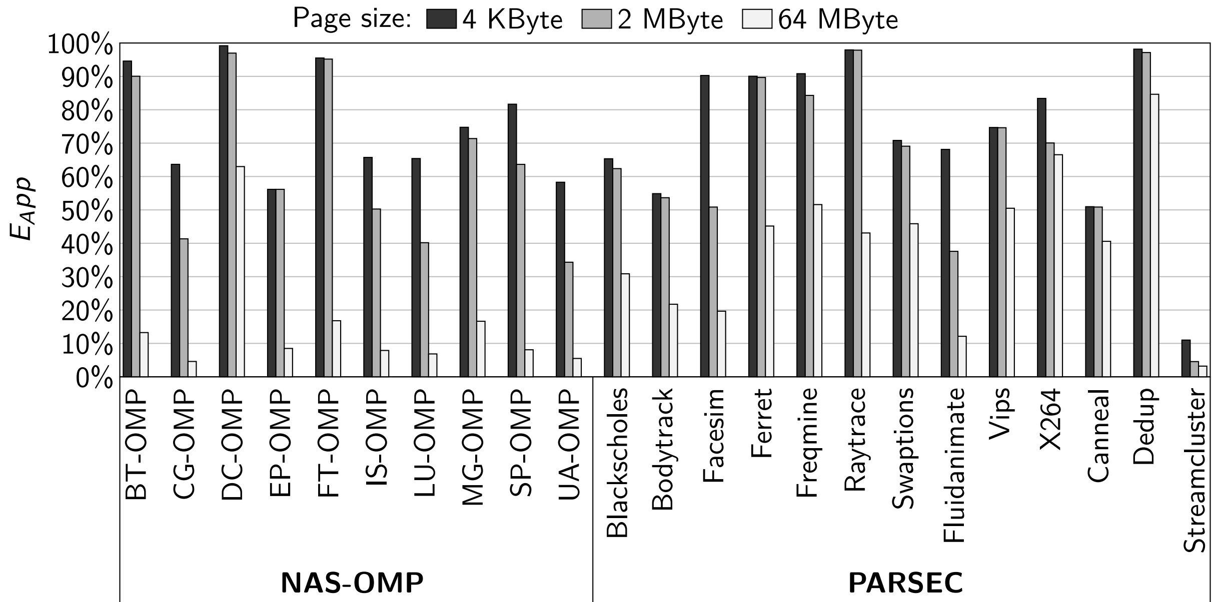


Figure 4.6: Application exclusivity E_{App} for 64 NUMA nodes and different page sizes.

with an almost equal number of pages on each node. However, the memory accesses of UA-OMP are very imbalanced, with node 0 handling more than 68% of the total accesses.

This intuition is confirmed by the values of the balance metrics, B_{Pages} and B_{Acc}, which are shown in Figures 4.8 and 4.9, respectively. In the figures, a value of 0 indicates perfect balance, while higher values indicate an imbalance in the distribution of pages and memory accesses between NUMA nodes. The maximum imbalance, when a single node stores all pages or handles all memory access, results in a value of 300 in this configuration. The RoundRobin policy is not shown in the figures, as its results were almost equal to the Interleave mapping.

Regarding the page balance, Random and Interleave are almost perfectly balanced, as expected. For the NAS-OMP benchmarks, only Remote and First-touch are significantly imbalanced. On the other hand, most PARSEC benchmarks are imbalanced with all policies except Random and Interleave. Regarding the memory access balance, most policies result in higher imbalances. Only the Balance policy results in a high balance for most of the benchmarks. It is important to mention that the Locality policy already results in a much better memory access balance than the First-touch policy. For example, the memory access balance of UA-OMP increases from 47% with First-touch to 95% with Locality. The Mixed policy results in a balance that is between the interleave and locality policies, as expected. Table 4.4 shows the average values of both metrics over all benchmarks for each policy.

We show a comparison of the B_{Acc} metric of the first-touch policy compared to the main improved mapping policies (Interleave, Locality, Balance, and Mixed) in Figure 4.10. The values were calculated for each application and policy with the following equation:

$$\Delta B_{Acc} = Policy_{B_{Acc}} - Firsttouch_{B_{Acc}} \tag{4.17}$$

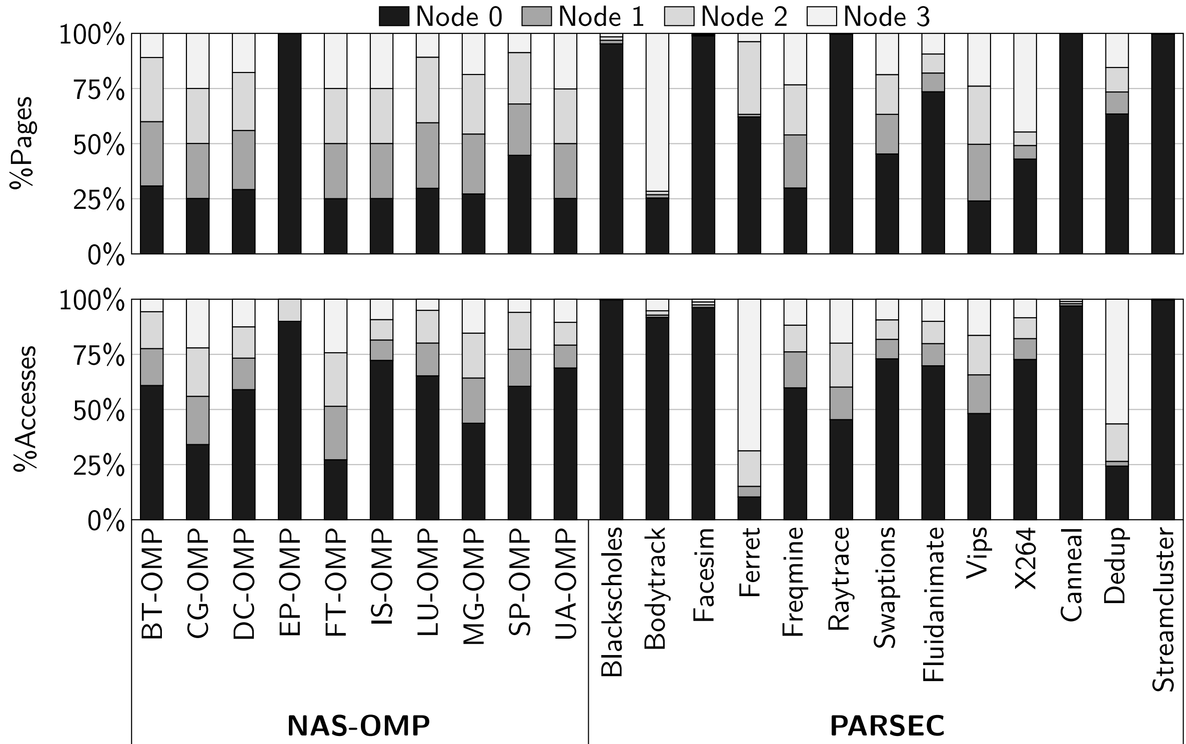


Figure 4.7: Page and memory access balance of the first-touch policy for a system consisting of 4 NUMA nodes. The color of each bar indicates how many pages or memory accesses are handled by each node.

The results show that for almost all of the benchmarks, each of the improved mapping policies results in a better balance than first-touch. Only a few of the benchmarks, such as CG-OMP and FT-OMP, are already well-balanced with first-touch and can not profit much from another policy. Most other benchmarks can benefit much more. It is interesting to note that the Balance policy, which results in the highest balance improvements in most cases since it focuses on the memory access balance, shows very similar results to the other policies. This indicates that this policy might not be necessary to achieve a better balance.

4.4.1.3 Locality

To compare different mapping policies in terms of their memory access locality, we evaluate the Loc_{Page} and Loc_{App} metrics defined in Section 4.2.5. Figure 4.11 shows these metrics for the first-touch page mapping for 4 KByte pages and 4 NUMA nodes. The results for the per-page locality Loc_{Page} show that the first-touch mapping results in a high page locality for most applications. However, when comparing it to the weighed

Table 4.4: Average values of the balance metrics of the mappings policies. Lower values indicate a more balanced behavior.

Metric	First-touch	Locality	Remote	RoundRobin	Interleave	Random	Mixed	Balance
B_{Pages}	121.2	73.7	177.2	0.007	0.438	1.01	51.1	60.6
B_{Acc}	171.2	66.4	163.8	28.9	30.2	32.1	48.2	12.2

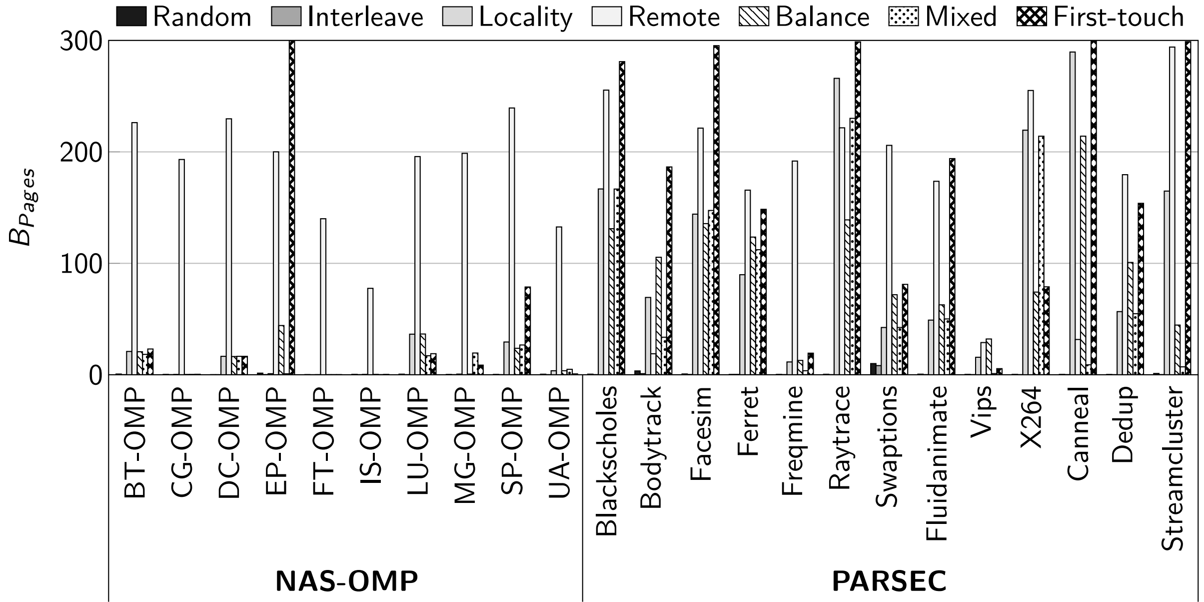


Figure 4.8: Page balance B_{Pages} of the data mapping policies. A value of 0 indicates perfect balance. Higher values indicate higher imbalance. The RoundRobin policy not shown in the figure has results almost equal to the Interleave policy.

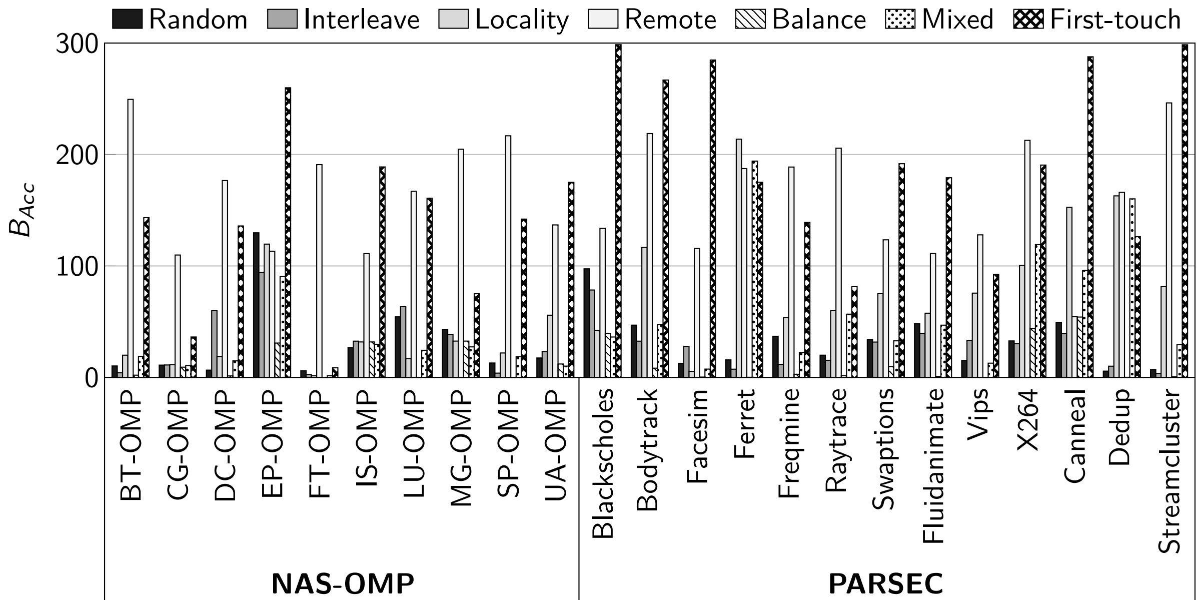


Figure 4.9: Memory access balance B_{Acc} of the data mapping policies. A value of 0 indicates perfect balance. Higher values indicate higher imbalance. The RoundRobin policy not shown in the figure has results almost equal to the Interleave policy.

locality Loc_{App}, the locality falls for most applications. This indicates that although the large majority of pages are placed to the NUMA node with the most accesses to them, other pages that are *not* placed on their local node receive an above-average number of memory accesses.

As an example of this behavior, consider the UA-OMP benchmark. Although more than 92% of its pages are placed on the NUMA node with the most accesses to them, these pages receive only 68% of the total memory accesses. The 8% of pages that are mapped

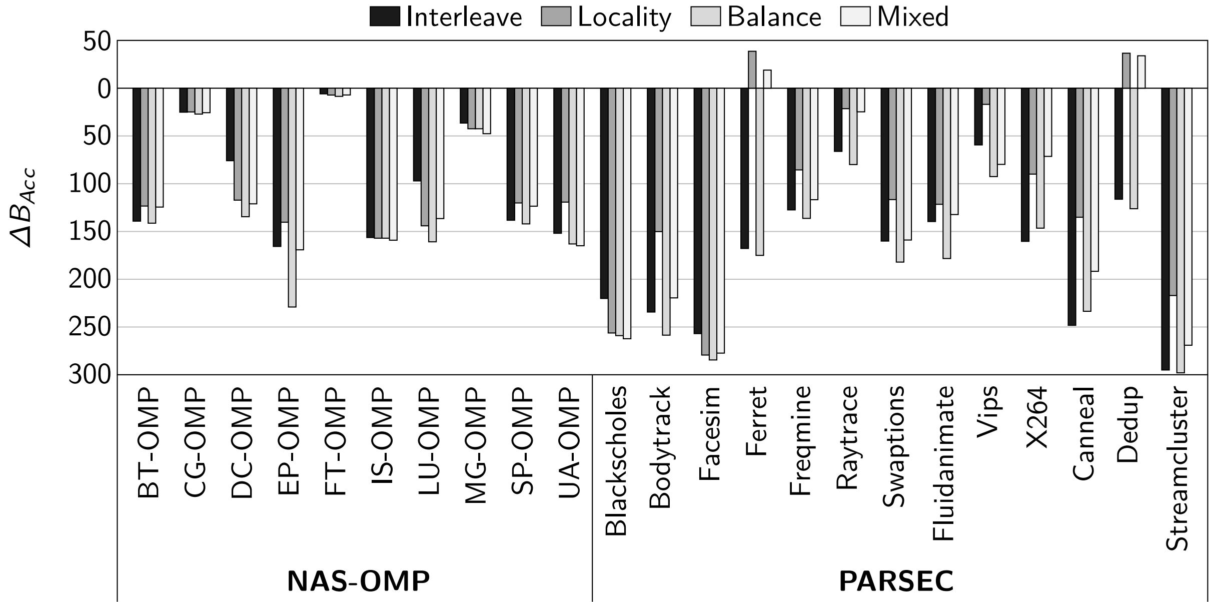


Figure 4.10: Comparing memory access balance between the main mapping policies. Lower values are better. Results are normalized to the first-touch policy (=0).

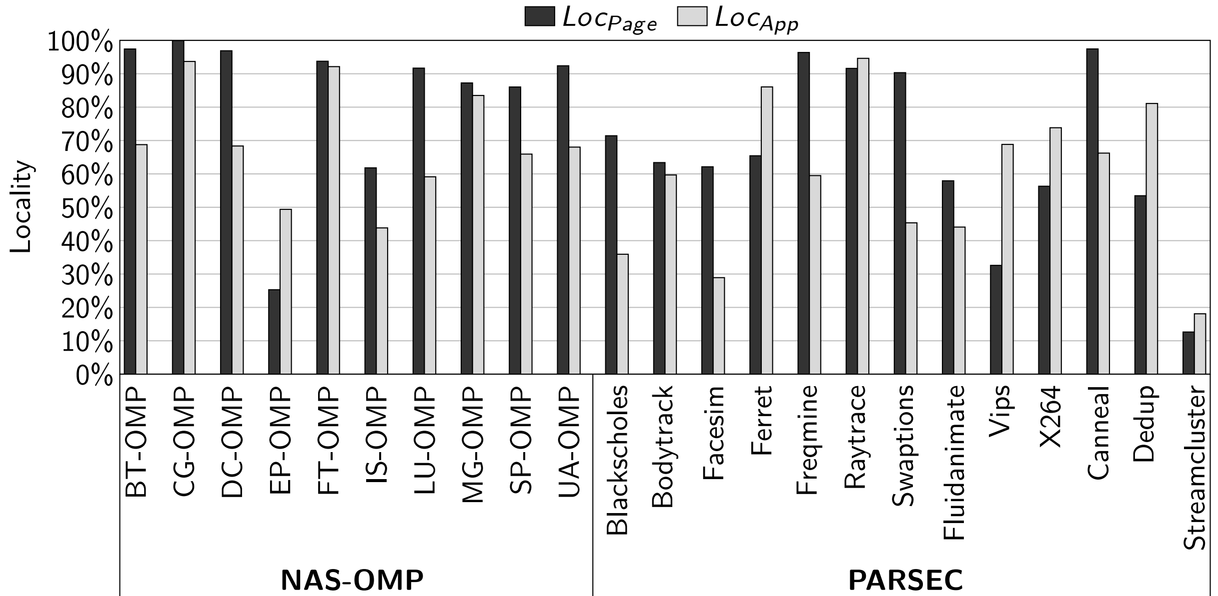


Figure 4.11: Locality of the first-touch page mapping for 4 KByte pages and 4 NUMA nodes.

to a non-local node receive 32% of memory accesses. Therefore, in order to perform a locality-based data mapping, it is possible to start with a first-touch mapping and then migrate a relatively low number of pages to their local NUMA nodes. These pages have an above average number of memory accesses for most of the benchmarks.

We show a comparison of the Loc_{App} metric of the first-touch policy compared

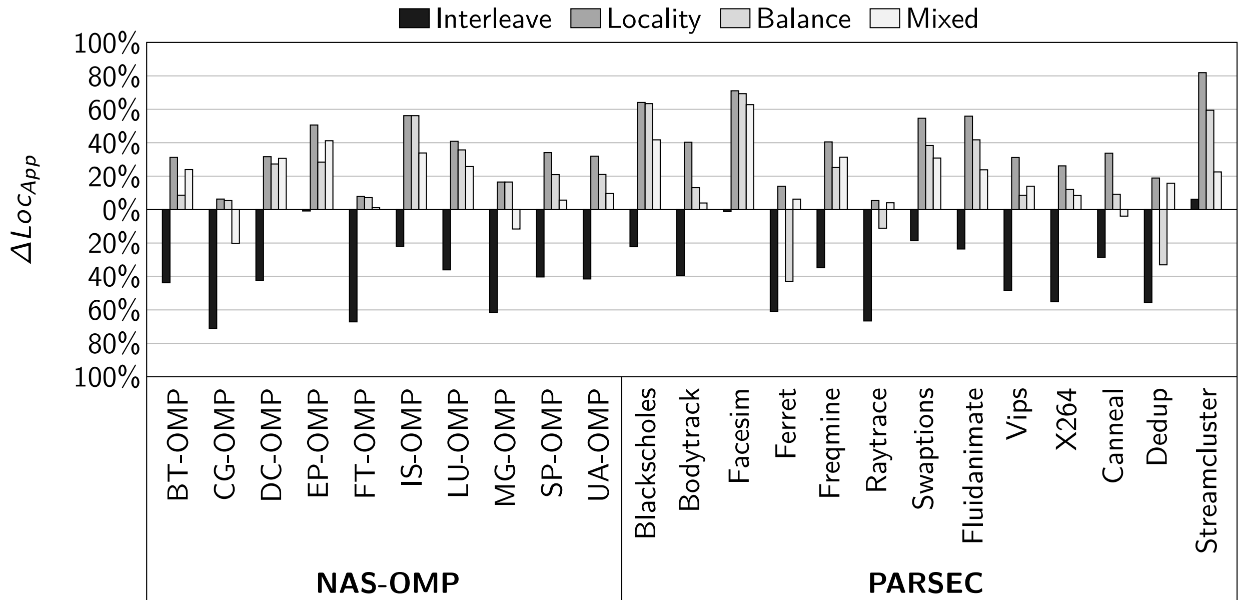


Figure 4.12: Comparing memory access locality between the main mapping policies. Values are normalized to the first-touch policy (=0%).

to the main improved mapping policies (Interleave, Locality, Balance, and Mixed) in Figure 4.12. The values were calculated for each application with the following equation:

$$\Delta Loc_{App} = Policy_{Loc_{App}} - Firsttouch_{Loc_{App}} \tag{4.18}$$

The results show that the Interleave policy greatly reduces Loc_{App}, causing a lower memory access locality, despite resulting in a better balance. The other three mapping policies consistently increase the memory access locality for almost all benchmarks, with the highest improvements for the Locality policy, as expected. This value, ΔLoc_{App}, can be used to estimate the performance improvements that can be gained from locality-based mapping policies.

4.4.2 Dynamic Behavior

To evaluate the dynamic behavior of the workloads, we measure the number of page migrations that have to be performed in order to maintain each page on the NUMA node with the most accesses during time windows of 10 ms. Figure 4.13 shows the migrations for all the workloads, considering no task migrations and an initial first-touch policy. As before, we assume that the system contains 4 NUMA nodes. The results show that most of the pages have to be migrated only once or not at all. Only Vips has a significant number of pages that need to be migrated twice or more. This reinforces our previous statement that a first-touch policy is a good starting point for page mapping and results in few page migrations. We did not notice a change in the of the page access balance during execution.

The values of the $Page_{dyn}$ metric are shown in Table 4.5, presenting a summary of the number of pages that have to be migrated for each second of execution time of the

benchmarks. Although the numbers for some benchmarks appear to suggest that a large amount of data needs to be migrated, the absolute amount of data is actually quite low. For example, for the Vips benchmark, which has the highest dynamicity, about 34,000 pages need to be migrated per second. This corresponds however to only about 130 MByte of data that needs to be copied between NUMA nodes, which is a reasonably low amount of data, even when considering an extremely short time window of 10 ms.

The dynamic page usage behavior of the SP-OMP benchmark is shown in Figure 4.14. For each time slice, we show the number of pages that need to be migrated to the NUMA node with the highest number of accesses during that slice, as well as the application exclusivity E_{App}. The results show that during the initialization of the application, the exclusivity varies between 60% and 80%, indicating shared accesses to memory pages.

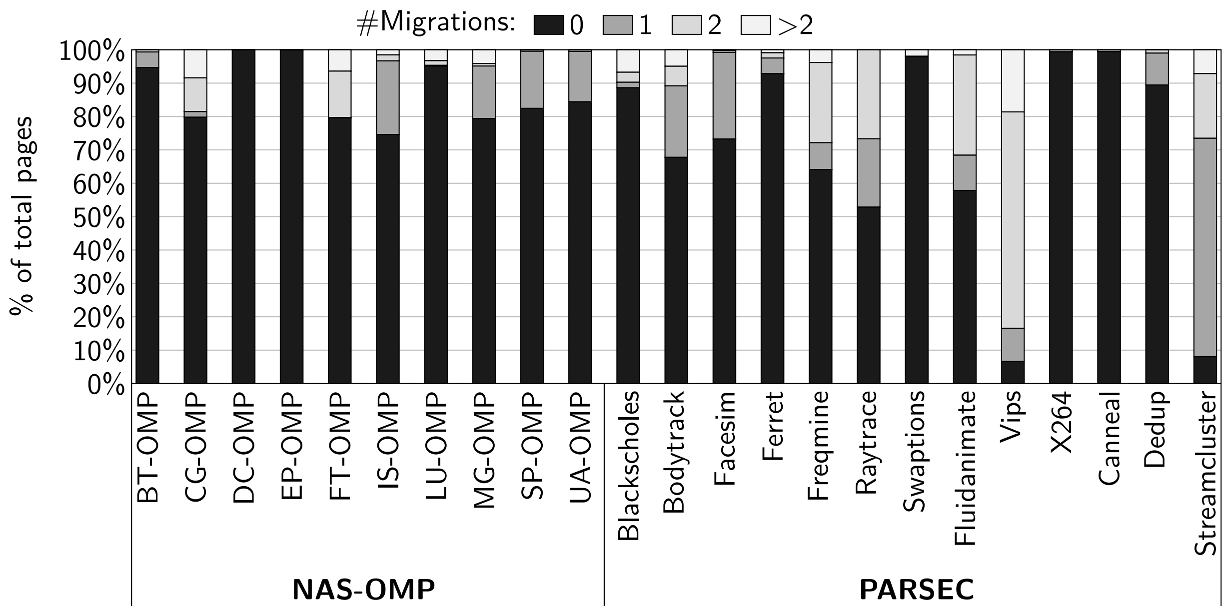


Figure 4.13: Number of page migrations.

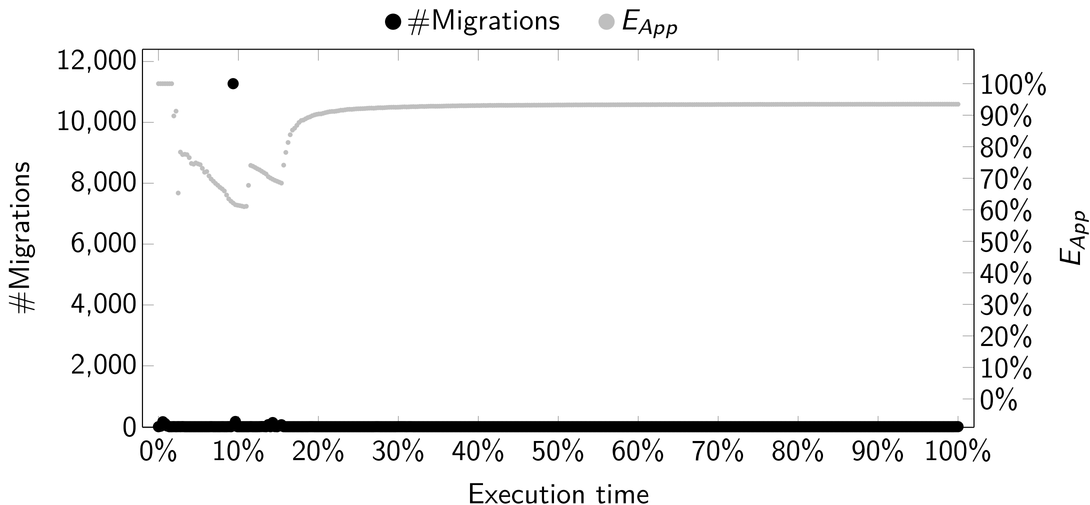


Figure 4.14: Dynamic page usage behavior of SP-OMP.

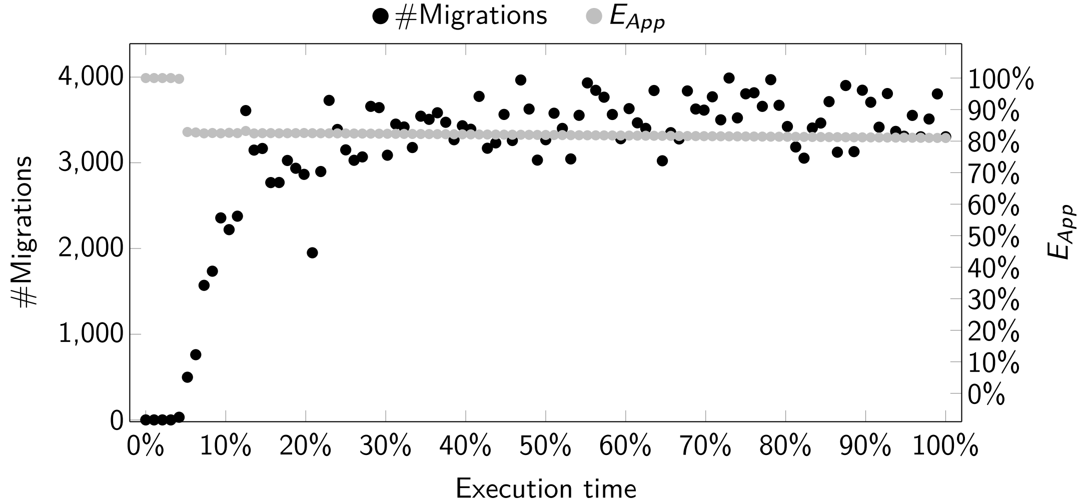


Figure 4.15: Dynamic page usage behavior of Vips.

Table 4.5: Dynamic page usage $Page_{dyn}$ of the benchmarks. Number of page migrations per second execution time.

	NAS-OMP										PARSEC												
Metric	BT-OMP	CG-OMP	DC-OMP	EP-OMP	FT-OMP	IS-OMP	LU-OMP	MG-OMP	SP-OMP	UA-OMP	Blackscholes	Bodytrack	Facesim	Ferret	Freqmine	Raytrace	Swaptions	Fluidanimate	Vips	X264	Canneal	Dedup	Streamcluster
$Page_{dyn}$	30	501	1	1	1,228	1,655	2419	2,348	31	221	4	20	104	85	915	5,019	1	4,709	34,124	52	9	123	2,792

As soon as the parallel part of SP-OMP starts, the exclusivity begins to stabilize and soon reaches its maximum of 92%. For the number of migrations, we can confirm that a significant number of page migrations are necessary only during the time slice at the start of the parallel computation. For the rest of the execution, the page usage is stable and almost no pages need to be migrated.

As an example of an application with a more dynamic behavior, Figure 4.15 presents the page usage during the execution of the Vips benchmark, which is the application with the most migrations in our experiments. The initialization phase with only a single thread lasts until about 5% of the total execution time. During the whole parallel phase of the application, the exclusivity stays at about 81%. However, in contrast to SP-OMP, pages need to be migrated during the whole execution to maintain them on the node with the highest number of accesses. At each time step, about 3,500 pages need to be migrated, which represents 0.4% of the total number of pages that Vips uses. With this very dynamic behavior, we expect fewer benefits from a data mapping policy.

Regarding the classification of dynamic memory access behavior presented in Section 1.3.1, our experiments showed that page usage remains the same between executions as long as neither input data nor the number of threads is modified. However, in contrast

to the communication behavior, memory addresses do change between executions of some applications, even when switching off Address Space Layout Randomization (ASLR), due to dynamic memory allocation during execution where addresses depend on the order in which allocation functions such as `malloc()` or `new()`. For this reason, a data mapping mechanism based on memory access traces can not be 100% accurate, as addresses of some pages might change between executions. On the other hand, page usage is much less dynamic *during* execution compared to the communication behavior. For example, even though pages get allocated dynamically, their access pattern does not change after allocation.

4.5 Summary of Page Usage

In this chapter, we introduced metrics and a methodology to analyze the page usage of parallel applications in order to determine their suitability for data mapping. To summarize the page usage behavior of the evaluated benchmarks, we saw that most of them are suitable for locality-based data mapping due to their high exclusivity. Even when increasing the page size from the default 4 KByte to 2 MByte and 64 MByte, the exclusivity reduces only slightly when considering that the number of threads is higher than the number of nodes. Compared to a first-touch mapping, only a few pages need to be migrated in order to achieve a mapping with maximum locality (less than 22% of pages on average). However, these pages represent a much higher percentage of memory accesses (more than 35% of total memory accesses on average), showing the importance of data mapping. Furthermore, most pages need to be migrated only once, which limits the overhead of migrations.

Regarding the balance of memory access, we can confirm that a policy that improves locality also improves balance in most cases. Nevertheless, a policy that focuses mostly on

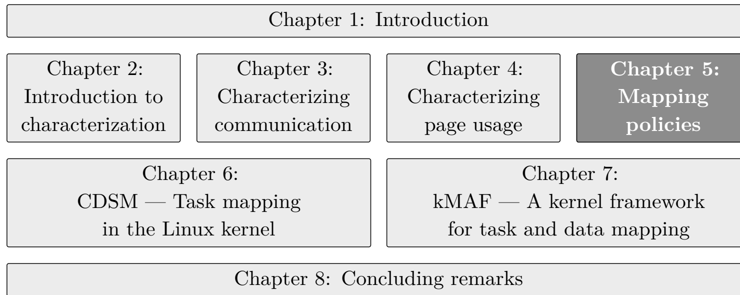
Table 4.6: Overview of benchmark suitability for data mapping. Darker cells indicate a higher suitability for mapping for the specified parameter.

	NAS-OMP										**PARSEC**												
Metric	BT-OMP	CG-OMP	DC-OMP	EP-OMP	FT-OMP	IS-OMP	LU-OMP	MG-OMP	SP-OMP	UA-OMP	Blackscholes	Bodytrack	Facesim	Ferret	Freqmine	Raytrace	Swaptions	Fluidanimate	Vips	X264	Canneal	Dedup	Streamcluster
Exclusivity E_{App}																							
Locality ΔLoc_{App}																							
Balance ΔB_{Acc}																							
Memory usage																							
Dynamic behavior $Page_{Dyn}$																							
Locality	✓	✓	✓		✓	✓		✓	✓	✓	✓	✓	✓					✓	✓	✓	✓	✓	✓
Balance	✓	✓	✓			✓		✓	✓	✓	✓	✓	✓	✓		✓		✓		✓	✓	✓	✓

balance increases memory access balance the most. As the default data mapping of the OS is based on first-touch, we expect benchmarks with a low first-touch locality to have higher improvements. Similarly, a low number of migrations indicates a lower overhead and therefore higher improvements. As mentioned before, the memory usage of EP-OMP and Swaptions is so low that few improvements from data mapping can be expected on hardware architectures whose cache size is similar to the total memory consumption of these applications.

Table 4.6 contains a summary of the benchmark behavior that was analyzed in this chapter. We show the main parameters that influence locality-based and balance-based data mapping policies. Darker cells indicate that a benchmark is more suitable for the given parameter. We expect higher performance and energy efficiency improvements for benchmarks that have a higher suitability for a particular policy. In Section 5.3, we will evaluate the performance improvements of various data mapping policies and compare them to the characterization presented in this chapter.

5 TASK AND DATA MAPPING POLICIES



Based on the characterization of the communication and page usage behavior of the parallel applications in the previous two chapters, we now evaluate the performance and energy impact of various task and data mapping policies. For the development of automatic mechanisms, such as those that will be presented in Chapters 6 and 7, this chapter makes the following contributions: (1) Measure the impact of task and data mapping and explain the reasons for improvements using a microarchitecture simulator. (2) Evaluate which mapping policies are most beneficial for parallel application performance on real machines. (3) Help find parameters for the automatic mechanism, such as the specific mapping algorithms.

In this chapter, we first verify the importance of task and data mapping with a microarchitecture simulator, evaluating both the performance and energy consumption improvements that can be achieved, as well as the interaction between task and data mapping. Then, we perform an in-depth analysis of various task and data mapping algorithms. Finally, we describe the Oracle mapping mechanism, which uses memory access traces to calculate an optimized task and data mapping for subsequent executions. It is therefore a semi-automatic mapping mechanism according to our classification in Section 1.3.2.

5.1 Evaluating Task and Data Mapping in a Simulator

Before we discuss how to generate specific thread and data mappings, we evaluate the impact of mapping on the hardware architecture. The goal of this section is to analyze *which* improvements can be achieved with improved mappings and how task and data mapping interact. Different combinations of mappings, generated based on memory access traces, are evaluated. Furthermore, we also perform an analysis of the energy consumption improvements due to mapping in the simulator, since accurate and detailed measurements of energy consumption are infeasible on current real hardware.

As mentioned in Section 1.2.2.2, mapping can reduce energy consumption for two reasons. Reducing execution time reduces static energy consumption (leakage) with the same proportion, since the processing units will be in a high power-consuming state for a shorter time. A shorter execution time does not directly affect the dynamic energy consumption, since the amount of work performed by the application (in terms of the number of instructions executed, for example) will remain about the same. However, reducing the number of cache misses and traffic on the interconnections reduces the dynamic energy consumption, leading to a more energy-efficient execution of parallel applications due

to improved mappings. For this reason, we expect overall energy consumption reductions that are lower than the execution time reductions.

5.1.1 Simulation Methodology

This section discusses the simulation environment, the benchmark used in the experiments and the mapping configurations.

5.1.1.1 Microarchitecture Simulator

We used SiNUCA (ALVES, 2014; ALVES et al., 2015), a performance and energy consumption validated, cycle-accurate microarchitecture simulator for the Intel x86 architecture. It contains a detailed model of all architectural components and supports multi-core NUMA architectures. Table 5.1 presents the configuration parameters of the simulated machine. The processors are based on the Intel Harpertown architecture (INTEL, 2008). The machine contains 4 processors, each with its own memory controller, forming 4 NUMA nodes in total. The processors consist of two cores, with private L1 instruction and data caches, and a shared L2 cache. The energy consumption of each component was estimated by feeding statistics generated by SiNUCA to the McPAT energy modeling tool (LI et al., 2013). Inside McPAT, the cache memories are modeled with CACTI (THOZIYOOR et al., 2008).

5.1.1.2 Benchmark

For the evaluation, we selected the SP-OMP benchmark from the OpenMP implementation of the NAS Parallel Benchmarks (JIN; FRUMKIN; YAN, 1999), since it has a high sensitivity to data and thread mapping. SP-OMP was executed with 8 threads, since the simulated system consists of 8 cores in total, and input size W due to simulation time constraints. We generate memory access traces of SP-OMP with the Pin tool (LUK et al., 2005) presented in Section 2.3 to calculate the mappings.

5.1.1.3 Mapping Configurations

In order to understand the importance of thread and data mapping and their impact on the performance and energy consumption, four different mapping configurations were simulated. For each configuration, we select either a *local* or *remote* policy for the thread and data mapping. In the local thread mapping (T_L), threads that access shared data are mapped close to each other in the hierarchy, while they are placed far apart in the remote thread mapping (T_R). Similarly, in the local data mapping (D_L), each memory page is mapped to the NUMA node that performs the most accesses to the page, while in the remote data mapping (D_R) each page is mapped to the node with the fewest accesses. This results in the following four combinations of configurations that are evaluated: $\mathbf{T_R D_R}$, $\mathbf{T_R D_L}$, $\mathbf{T_L D_R}$ and $\mathbf{T_L D_L}$.

Table 5.1: Configuration of the machine that was simulated in SiNUCA.

Property	Value
System	4×2-core processors, L1I/L1D cache per core, L2 caches shared between 2 cores
NUMA	4 memory ctrl./NUMA nodes, NUMA factor: 3, min/max latency L2 to memory ctrl.: 32/96 cycles
Execution cores	OoO, 1.8 GHz, 65 nm, 12 stages, 16 B fetch size, 96-entry ROB, PAs branch predictor
L1I/L1D caches	32 KB, 8-way, 64 B line size, LRU policy, 1 cycle, MOESI protocol, stride + next-line prefetch
Shared L2 caches	512 KB, 8-way, 64 B line size, LRU policy, 4 cycles, stream prefetch
Interconnection	Bi-directional ring, hop latency: 32 cycles
DRAM	DDR2 667 MHz (5-5-5), 8 DRAM banks/channel, 2 channels, 8 KB row buffer

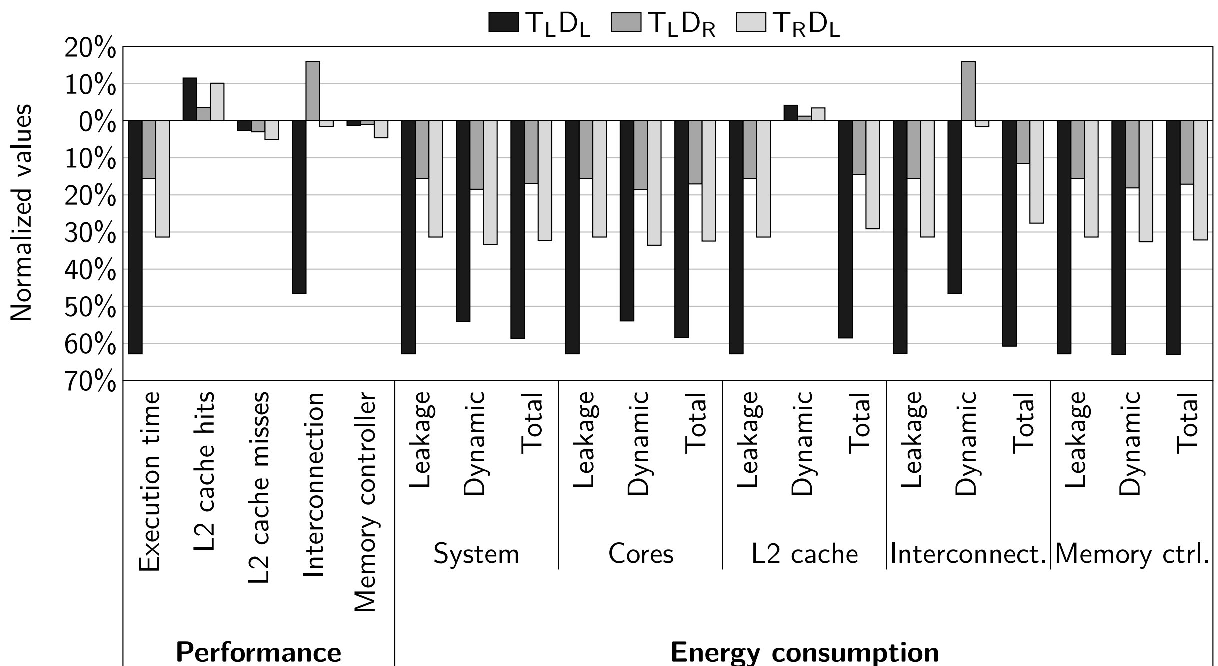


Figure 5.1: Performance and energy consumption of the SP-OMP benchmark in SiNUCA, normalized to the remote thread and data mapping (T_RD_R).

5.1.2 Simulation Results

Figure 5.1 presents the simulation results for the system components. The results are normalized to the values of the T_RD_R mapping, which had the lowest performance. We begin with an analysis of the performance results of the different mappings, followed by a discussion of the energy efficiency improvements.

5.1.2.1 Performance

We observe that using only the local data mapping (T_RD_L) improves the performance by 31%, providing higher benefits than performing only the local thread mapping (T_LD_R), which reduced execution time by 15%. The local data mapping (T_RD_L) improves the accesses to the main memory, by reducing the distance between the threads and the data they access. Since these accesses are off-chip, they are high latency operations. On the other hand, the local thread mapping (T_LD_R) improves the usage of the shared cache memories (L2) by reducing the competition for the L2 cache space and improving the cache hit and miss ratios. As these are low-latency on-chip accesses, this mapping results in fewer improvements compared to the local data mapping.

When combining both local mappings (T_LD_L), the application performance improved by 62%, which is higher than the sum of the results from the local thread and data mappings applied separately. T_LD_L is able to reduce the number of cache misses and cache-to-cache transfers due to the local thread mapping, and also reduces the main memory average latency and interconnection contention due to the local data mapping. In addition, the local thread mapping also increases the gains from the local data mapping for pages that are shared between threads. By mapping threads that share data to the same NUMA node, main memory accesses from these threads will be considered as local, augmenting the benefits of the local data mapping (DIENER et al., 2014).

5.1.2.2 Energy Consumption

As discussed in Section 1.2.2.2, thread and data mapping can also improve the energy efficiency of parallel applications. For all the evaluated mappings, the more efficient execution also reduced energy consumption. Leakage was reduced for all components, correlating with the execution time as expected. The main sources of dynamic energy savings were the reduction on the number of L1 cache misses, the interconnection traffic reduction and the reduction in the number of main memory accesses, since most of the accesses are being treated by the L2 cache memory. The overall energy reductions were slightly lower than the execution time reductions due to lower dynamic energy savings.

5.1.3 Summary

This section showed that substantial performance and energy improvements can be accomplished with mapping. The main benefits from the local thread and local data mapping can be observed on the memory sub-system and the interconnections, enabling a more efficient usage of resources. By combining both types of mappings, a compounding effect is achieved, resulting in higher improvements than when applying each technique separately.

5.2 Evaluating Task Mapping Policies

In this section, we evaluate the performance improvements that are achieved by the task mapping policies proposed in Section 3.3 on a real machine.

5.2.1 Experimental Methodology

This section contains methodological considerations regarding the performance experiments.

5.2.1.1 Benchmarks and Evaluation System

We performed the experiments on our main evaluation system, *Xeon*, which contains 64 processing units in total, divided into 4 NUMA nodes. Task mapping results on the other two systems are qualitatively similar, and we do not present them here. Detailed information about the configuration of the system is shown in Section 2.2. We selected the NAS-OMP (with the *A* and *B* input sizes) and PARSEC (*native* input size) benchmark suites since their results are representative for the other suites as well. All applications were executed with 64 threads. We evaluate all mapping policies that were presented in Section 3.3. In all policies except OS, no migrations during execution were performed.

5.2.1.2 Statistical Evaluation

The presentation of performance and energy consumption results in this chapter and the rest of the thesis is based on the following methodology. All experiments are executed at least 10 times for all configurations. Statistics are collected regarding each execution (such as execution time, energy consumption, L3 cache misses and QPI traffic) and we calculate the average of each type of statistic. We then normalize all values to our baseline, the results of the OS.

Performance for each benchmark is presented in terms of *improvements* compared to the baseline, calculated with Equation 5.1.

$$mean_{Performance} = \left(\frac{average\ execution\ time\ with\ OS}{average\ execution\ time\ with\ mechanism} - 1 \right) \times 100\% \qquad (5.1)$$

For the other statistics, we present results as a *reduction* compared to the baseline, described by Equation 5.2.

$$mean_{Others} = \left(1 - \frac{average\ value\ with\ mechanism}{average\ value\ with\ OS} \right) \times -100\% \qquad (5.2)$$

The accuracy of the presented averages is shown with a confidence interval for a confidence level of 95%, assuming a Student's t-distribution.

To compare the results of different mechanisms for multiple benchmarks, we use the geometric mean of the benchmark results, since the normalized results represent ratios.

However, some of the ratios might be negative, if a mechanism causes a performance reduction compared to the OS, for example. Negative values are not directly supported by the geometric mean. For this reason, we calculate the mean in a slightly modified way, by adding 100% to each value and subtracting this value again after the mean is calculated. This operation is shown in Equation 5.3, where x_i represents the gain or reduction of benchmark i compared to the baseline, and N is the number of results for which to calculate the mean.

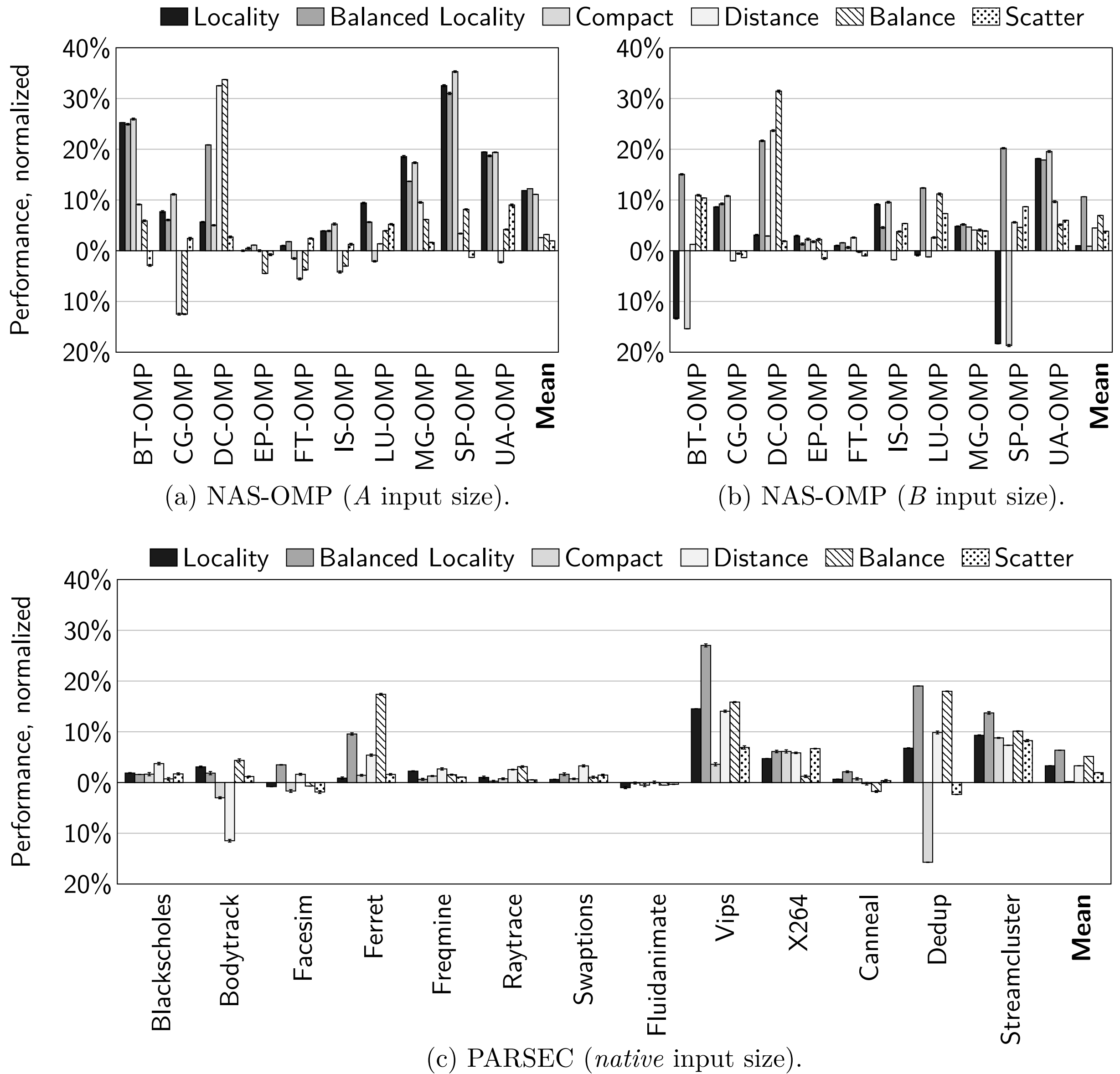
$$geomean\,(x_i) = \left(\prod_{i=1}^{N}(x_i + 100\%)\right)^{1/N} - 100\% \tag{5.3}$$

5.2.2 Results

Figure 5.2 shows the performance gains compared to the OS mapping. For NAS-OMP with the A input, most benchmarks profit from the Locality policy, as predicted by our analysis. With the A input, this policy never reduces performance. The Balanced Locality policy has similar results as Locality for all benchmarks except DC-OMP. For the benchmarks that have a nearest neighbor communication pattern, the Compact policy improves performance, but reduces it in some cases, such as LU-OMP. The Distance and Balance policies only show performance improvements close to the Locality policy for the DC-OMP benchmark, which benefits from a better balance due to its low communication ratio. The Scatter policy never results in significant performance gains and reduces it in many cases. On average, the Locality, Balanced Locality, and Compact policies show improvements of more than 12%, the other policies gain less than 4%.

For the NAS-OMP benchmarks with the B input, the Locality policy reduces performance for the benchmarks that are imbalanced (BT-OMP, LU-OMP, and SP-OMP). On the other hand, Balanced Locality achieves the highest gains, proving that only taking locality into account is not sufficient for applications with this characteristic. The Load Balance policy (not shown in the figure) has improvements of less than 5% for the 3 benchmarks, indicating that balancing the load is not as effective as balancing the communication in these cases. The other benchmarks show a similar behavior as the A input, with lower average gains. This echoes our discussion of the communication ratio, were we expected lower improvements when the ratio decreases. On average, Balanced Locality achieved the highest improvements, of 10.9%. As several benchmarks benefit from balancing, the Balance policy has the second-highest improvements, of 7.3%. The other policies gain less than 5%.

As discussed in Section 3.4, the PARSEC benchmarks generally have lower metrics for task mapping than the NAS-OMP benchmarks, which is reflected in the performance results. Five benchmarks (Ferret, Vips, X264, Dedup, and Streamcluster) benefit from communication-aware thread mapping. Most of them benefit from both the Locality and the Balance policies, but the Balanced Locality policy, which combines both, results in the highest improvements in most cases. The Compact, Distance, and Scatter policies do not improve performance consistently and result in performance losses in several cases.



(a) NAS-OMP (*A* input size).

(b) NAS-OMP (*B* input size).

(c) PARSEC (*native* input size).

Figure 5.2: Performance improvements of different task mapping policies on the *Xeon* machine, compared to the OS mapping.

On average, Balanced Locality has again the highest gains of 6.7%, followed by Balance (5.4%) and Locality (3.4%).

5.2.3 Comparison to Predicted Task Mapping Suitability

Section 3.5 summarized our benchmark characterization regarding the suitability for task mapping. All NAS-OMP benchmarks except EP-OMP and FT-OMP were characterized as suitable for at least one type of task mapping. This is reflected well in the performance results. As predicted, EP-OMP and FT-OMP had almost no performance improvements. BT-OMP, LU-OMP, and SP-OMP with the *B* input can not benefit from locality-based policies due to their imbalance, but have high gains from balance policies.

The other NAS-OMP benchmarks have substantial performance improvements from both policy types.

The PARSEC benchmarks were characterized as generally less suitable for task mapping, with Ferret, Dedup, and Streamcluster predicted to benefit from both types of task mapping. These benchmarks also have substantial performance gains. Two other benchmarks Vips and X264, also have high gains, in contrast to the predictions. We assume that these gains are mostly due to the prevention of task migrations during execution, which can have a substantial impact on performance as these applications have a highly dynamic behavior. Vips is based on a thread pool, where tasks that will be calculated are put in a list. X264 creates 1,024 threads, which only exist for a short time. All other PARSEC benchmarks have only negligible performance differences with task mapping, as correctly predicted by our characterization.

Although the general suitability for task mapping was characterized with a high accuracy, the amount of performance improvements can not be predicted accurately in all cases with our model. For example, CG-OMP and DC-OMP with the B input size had a very similar suitability for locality-based task mapping, but CG-OMP has much higher performance improvements from such a policy. To achieve such a high level of precision, a more detailed model of the application and hardware architecture is necessary, such as information about the exact timing of memory accesses, interconnection latencies and cache sizes. This requires a highly-detailed architectural simulator, which would limit the characterization to small benchmarks.

5.2.4 Summary

From the results presented in this section, we conclude that increasing locality is the most important way to perform communication-aware thread mapping for most parallel applications. However, some applications can benefit from improving the balance of the communication, achieving higher performance gains and avoiding the performance reduction that a locality-based policy can cause. Simple mapping policies that do not take the communication behavior into account only improve performance in some cases and provide no consistent improvements over the OS.

5.3 Evaluating Data Mapping Policies

This section evaluates the performance improvements of the data mapping policies that were proposed in Section 4.3.

5.3.1 Methodology

We evaluate the data mapping policies and compare them to the baseline, the first-touch policy of Linux. Since its results were very close to the Interleave policy, the RoundRobin mapping will not be shown in the figures. As these mapping policies are

calculated based on memory access traces, we require that the addresses themselves do not change between multiple executions. We disable Address Space Layout Randomization (ASLR) in the kernel, which makes most memory allocations reproducible between different executions. However, addresses can change in case the parallel application calls `malloc()` from multiple threads. Most NAS-OMP benchmarks and some PARSEC applications have static addresses between executions, and we filter out those applications where the memory addresses change.

Since the behavior on our evaluation systems *Itanium*, *Xeon*, and *Opteron* differ considerably, we present results for all three of them. Threads were pinned to the execution cores with a Compact thread mapping, to remove the influence of thread migrations during execution.

5.3.2 Performance Results

The results for the performance improvements (compared to the baseline, the first-touch policy of Linux) for the three evaluated architectures are shown in Figures 5.3, 5.4 and 5.5. For the Mixed policy, we show the improvements for a value of *minExcl* of 90%.

5.3.2.1 Itanium

Figure 5.3 shows the results for the *Itanium* machine. The highest improvements were achieved for the CG-OMP benchmark, of up to 67% for Locality, Balanced and Mixed. Due to *Itanium*'s relatively slow interconnection, the Interleave policy achieves generally lower improvements than the Locality and Mixed policies. Since it still takes locality into account, the Balanced policy achieves higher improvements than Interleave. As the machine has only a small number of NUMA nodes (2), even the Random policy improves performance in many cases, as it has a higher chance of mapping the page to the local node. The average performance improvements are: 5.2% with Random, 5.7% with Interleave, 20.9% with Locality, 14.4% with Balanced and 19.4% with Mixed. The fact that the improvements with Mixed are lower than the improvements with Locality (even when using a 99% minimum exclusivity for mixed) shows that on this machine, improving memory access locality is more important than balance.

5.3.2.2 Xeon

Figure 5.4 shows the results for the *Xeon* machine. Due to the higher number of threads and NUMA nodes, performance improvements from data mapping are higher than on *Itanium*, despite *Itanium*'s slower interconnection. For the same reason, the balance policies achieve improvements closer to the locality policies. The SP-OMP benchmark achieved the highest improvements, even for the Random policy, of up to 108%. For the other benchmarks, the Random policy can not reduce execution time significantly or even increases it (up to 37% for MG-OMP). On average, performance was improved by 8.7% with Random, 15.8% with Interleave, 18.5% with Locality, 14.6% with Balanced and 23.7% with Mixed. The results show that increasing only locality is not enough to achieve

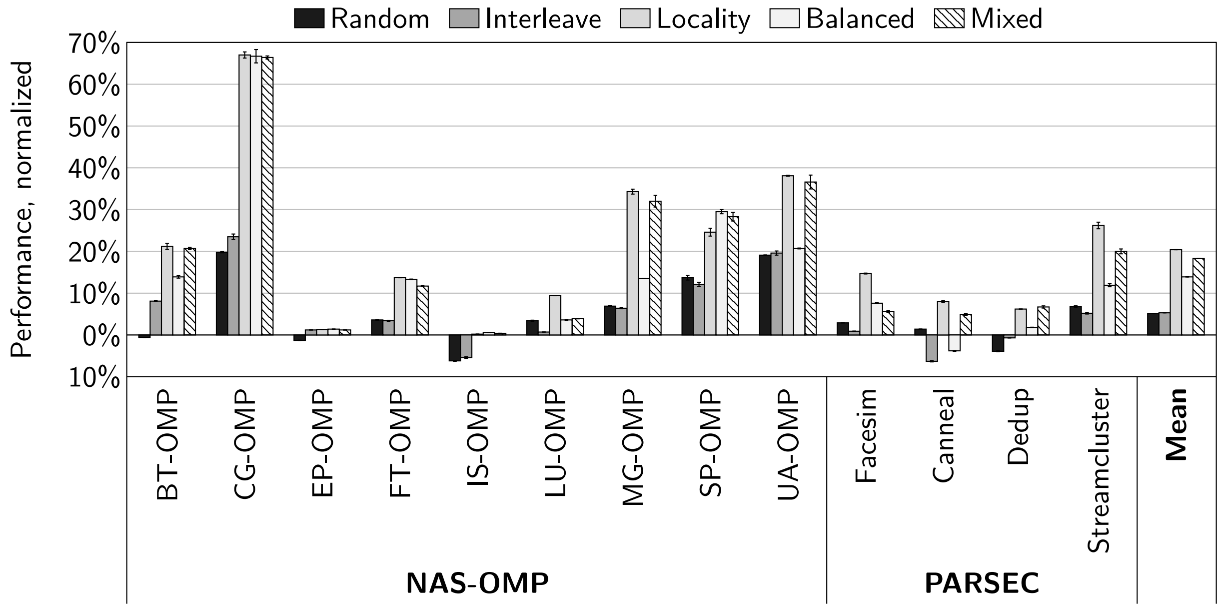


Figure 5.3: Performance improvements on *Itanium* with different data mapping policies, compared to the first-touch mapping.

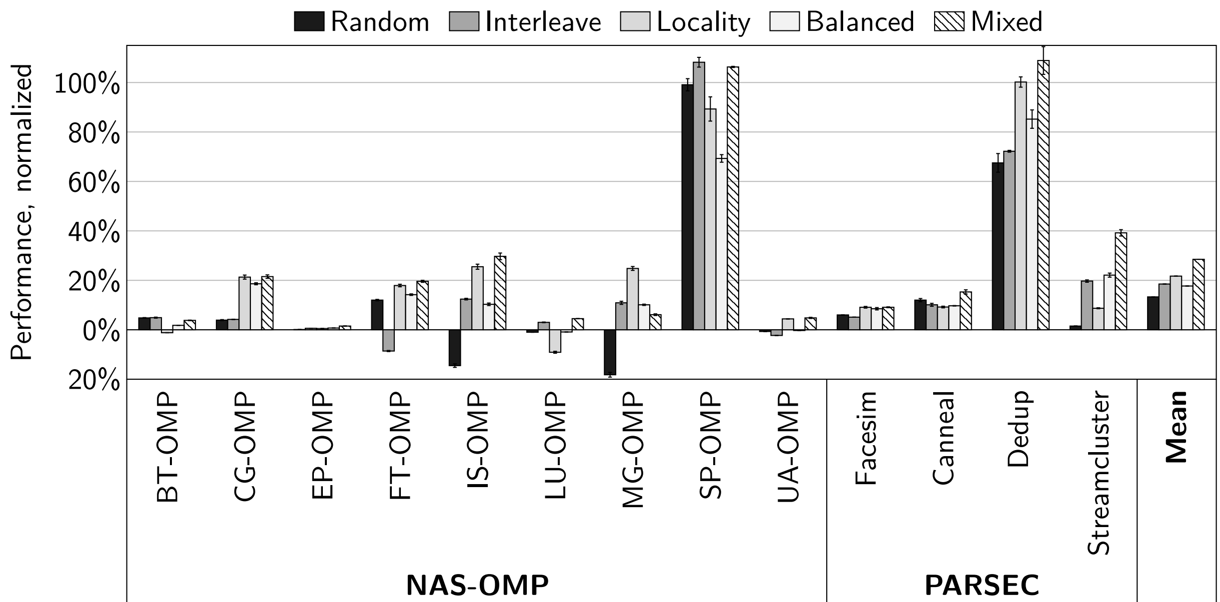


Figure 5.4: Performance improvements on *Xeon* with different data mapping policies, compared to the first-touch mapping.

optimal improvements. For several benchmarks (BT-OMP, LU-OMP, and Canneal), the Locality policy actually reduces performance compared to First-touch.

5.3.2.3 Opteron

Figure 5.5 shows the results for the *Opteron* machine. Due to its large number of NUMA nodes (8) and high NUMA factor, the highest improvements were achieved on

this machine. In many cases, the Random and Interleave policies reduce performance. Balancing memory accesses is still important though, as shown by the higher improvements for Mixed than for Locality. As before, the Locality policy increases execution time slightly compared to the baseline for some benchmarks (CG-OMP, LU-OMP, and UA-OMP). Similar to the Xeon machine, the highest improvements were achieved for SP-OMP, up to 272% with the mixed policy. On average, performance was improved by 6.3% with Random, 7.2% with Interleave, 50.0% with Locality, 23.9% with Balanced and 61.8% with Mixed.

5.3.3 Comparison to Predicted Data Mapping Suitability

In Section 4.5, we summarized our predicted suitability for data mapping, considering 64 threads and 4 NUMA nodes, which corresponds to our *Xeon* machine. Comparing the performance results on *Xeon* to the prediction showed that suitability was analyzed correctly in most cases. From the NAS-OMP benchmarks, we only characterized EP-OMP and LU-OMP as being unsuitable for data mapping, which is reflected in their very low improvements and performance reduction in some cases. All other NAS-OMP benchmarks improve performance with data mapping. The PARSEC benchmarks evaluated in this section were all characterized as suitable for data mapping, and they have significant performance gains.

Similar to the analysis for task mapping, although the characterization was very accurate in terms of which benchmarks can benefit from data mapping, the performance improvements still vary widely between benchmarks, which is not well reflected by the metrics. This is due to two main factors. First, not all information about the memory

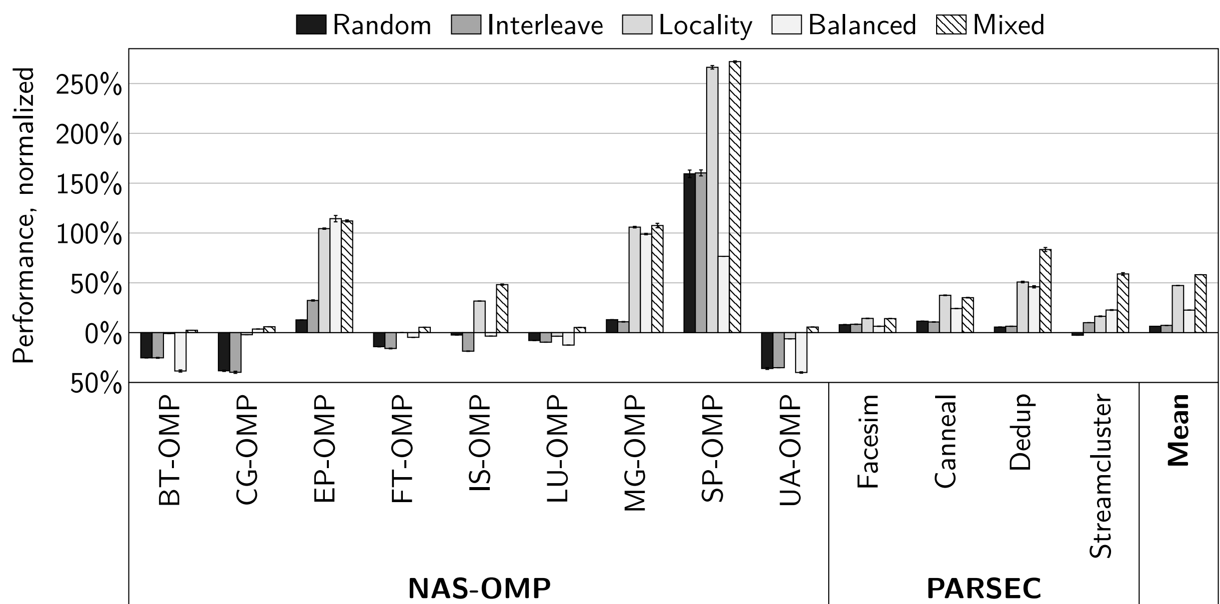


Figure 5.5: Performance improvements on *Opteron* with different data mapping policies, compared to the first-touch mapping.

access pattern of the application is collected. For example, the suitability of the pattern for the cache line prefetcher is not taken into account. If the prefetcher can perform an accurate prediction of future memory access, data mapping is less effective, as more memory accesses can be filtered by the caches. On the architectural level, cache size, interconnection speed and contention influences the benefits of data mapping. To provide such a level of precision, much more detailed architectural simulation is required, which would limit the applicability of the characterization to small applications.

5.3.4 Summary and Discussion

Several important conclusions can be drawn from these performance results.

First-touch often has a negative impact on performance. Compared to most other policies, the First-touch policy of modern operating systems often has a negative impact on the performance of parallel applications. In many cases, even a random assignment of pages to NUMA nodes outperforms first-touch. The reason for this behavior is that in many parallel applications, one thread initializes the data and forces page allocation on a single NUMA node, leading to an increased number of memory accesses to that node, while a random policy can balance the memory access load more equally among the nodes.

Locality is still more important than balance. Regarding the importance of locality and balance, results depend on the hardware architecture. On traditional NUMA systems with a relatively slow interconnection, such as our *Itanium* machine, memory access locality is the most important metric for performance improvements. For modern NUMA architectures, the importance of balancing the memory accesses between memory controllers is becoming increasingly important, although locality is still the most important metric to optimize, as evidenced by the results of the Interleave and Balanced policies compared to the Locality policy. An important reason for this result is that improving locality also improves balance for most applications.

Mixed policies can provide the highest improvements. Neither Locality nor Balanced are able to improve performance in all cases and actually reduce performance significantly in some instances. Taking both locality and balance into account (as done by our Mixed policy) when mapping pages to NUMA nodes achieves the highest results overall and also avoids the performance decrease of the other policies. In this way, highly exclusive pages can benefit from the locality, while shared pages are distributed among the nodes to balance the memory accesses between memory controllers.

5.4 An Oracle Mapping Mechanism

This section describes an optimized task and data mapping mechanism, which we refer to as the *Oracle* mechanism, consisting of a memory tracer and mapping policies.

To determine the memory access behavior, we adapted the Pin-based memory tracer presented in Section 2.3. The tool outputs the communication matrix and the page usage of the parallel application, and uses them to calculate the task and data mapping

policies. Despite the overhead caused by the tracing, it is still feasible to characterize very large applications, such as the ones we use in the thesis. Since we do not save the full memory access trace, but only the behavior, the data generated is very small (less than 100 MByte per application).

To calculate the task mapping from the communication matrix, we use either the Locality or Balanced Locality policy introduced in Section 3.3, whichever resulted in the highest performance gains for each application on each machine.

For the calculation of the data mapping, we apply data mapping policy that had the highest performance results for each application and machine to the global page usage. The mechanism outputs a file that contains a list of pages and the NUMA node where they should be mapped to.

We implemented the Oracle mechanism as a kernel extension for Linux. When a parallel application starts, the kernel reads the files that contain the thread and data mapping and maps tasks and pages according to the specified mapping when they are created or accessed for the first time, respectively. In this way, the Oracle mechanism is independent of the parallel API that the application uses. It is difficult to perform dynamic migrations according to a predefined policy with a high accuracy For this reason, we modify those parallel applications that have a dynamic communication behavior such that at each phase change, they perform a migration according to the calculated Oracle task mapping. For data mapping, only 1 application (Vips) has a significant dynamic page usage, but we did not find a reliable way to perform an Oracle data migration, and we maintain the data mapping static for the applications.

5.5 Summary

In this chapter, we evaluated a large set of task and data mapping policies according to their performance on a variety of parallel machines. We concluded that for most applications, increasing locality of memory accesses is the most important way to improve performance. However, some applications can also benefit from improving the balance. The results have shown that for most benchmarks, our behavior analysis in Chapters 3 and 4 were correct in predicting the improvements that are possible.

Based on these observations, we proposed a trace-based Oracle mechanism to calculate optimized task and data mappings. This mechanism serves as a baseline for the maximum improvements that can be achieved in the next chapters. The Oracle mechanism has complete access to the application's behavior and has no runtime overhead as all the mapping decisions are made before execution. However, it requires expensive memory tracing operations before the actual execution. In the next chapters, our goal is to introduce *automatic* mechanisms that achieve similar results as the Oracle, based on the task and data mapping policies presented here. These mechanisms have knowledge of only a part of the behavior and need to perform mapping decisions during execution.

Part 2: Automatic Mapping Mechanisms

Chapter 1: Introduction

Part 1

Chapter 2: Introduction to characterization

Chapter 3: Characterizing communication

Chapter 4: Characterizing page usage

Chapter 5: Mapping policies

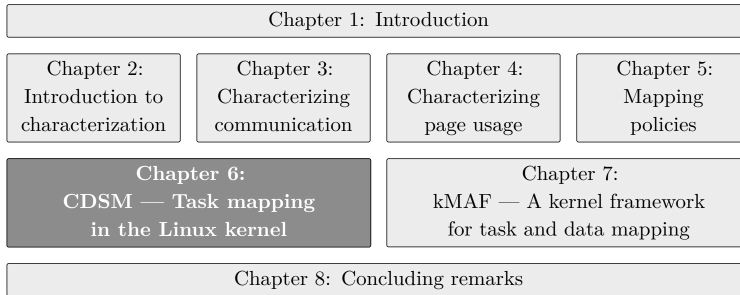
Part 2

Chapter 6: CDSM — Task mapping in the Linux kernel

Chapter 7: kMAF — A kernel framework for task and data mapping

Chapter 8: Concluding remarks

6 CDSM: COMMUNICATION-AWARE TASK MAPPING



In the first part of the thesis, we performed an extensive evaluation of the memory access behavior of parallel applications and developed mapping policies to improve it based on complete and precise information about the behavior. In the second part, we will present mapping mechanisms that perform the behavior analysis and mapping automatically during the execution of the application. In order to keep the runtime overhead manageable, these mechanisms need to deal with *incomplete* information, through the use of sampling, coarser detection granularities and mapping heuristics.

In this chapter, we introduce our first automatic mechanism, *Communication Detection in Shared Memory (CDSM)*, which is a kernel-based mechanism to perform communication-aware task mapping. We begin this chapter with a brief overview of the challenges of online mapping, followed by a review of previous work in the task mapping area. Then, we present CDSM and evaluate its performance and energy consumption improvements.

6.1 Introduction

CDSM is a mechanism to perform communication-aware task mapping in shared memory architectures. In contrast to the Oracle presented in the previous chapter, it is an automatic mechanism as per our classification in Section 1.3.2. Automatic mapping mechanisms face two main challenges, the runtime overhead and the limitation to past behavior. First, the detection of communication has to be performed accurately and with a low overhead. As applications generate a huge number of memory accesses, the detection can be done by sampling the memory accesses. Second, since automatic mechanisms have no knowledge about future behavior, it must be predicted using knowledge about the past behavior.

In this chapter, we will introduce CDSM, our mechanism to detect communication and use the detected information to perform an optimized task mapping. CDSM is based on the central idea of using page faults to sample and analyze memory access behavior, which generates a very low overhead for the running application. The detected behavior is used to optimize the mapping of tasks to cores. Since CDSM is implemented on the kernel level, it is independent of the parallel API that the application uses and requires no changes to the application or the hardware. It can handle multithreaded as well as multiprocess applications. We evaluated CDSM with the benchmarks presented in Section 2.1 and achieved substantial performance and energy consumption improvements with a negligible overhead.

6.2 Related Work on Task and Process Mapping

Several related mechanisms that focus on thread and process mapping considering the communication have been proposed. In this section, we analyze these mechanisms and compare them to our proposed approach. Specific algorithms that can be used to calculate a task mapping are not part of the thesis, but were discussed briefly in Section 3.3.

6.2.1 Applications Based on Message Passing

For parallel applications that communicate through explicit message passing, such as MPI, most previous research focuses on methods to trace the messages and use the information to perform a static process mapping. Mechanisms to trace MPI messages include the MPI Parallel Environment (MPE) (CHAN; GROPP; LUSK, 1998), eztrace (TRAHAY et al., 2011) and the Intel Trace Analyzer and Collector (INTEL, 2013a). MPIPP (CHEN et al., 2006) is a framework for static process mapping in MPI, consisting of a message tracer and a mapping algorithm.

Extensions for several MPI frameworks have been proposed to perform a mapping of processes to cores such that communicating processes are executing close to each other. In (MERCIER; JEANNOT, 2011), an extension for the MPICH2 framework (GROPP, 2002) is proposed, where the communication pattern can be provided by the application developer or generated via execution traces. In (HURSEY; SQUYRES; DONTJE, 2011), a similar extension for Open MPI (GABRIEL et al., 2004) is introduced, where the user needs to specify the desired mapping. Both frameworks focus on static mappings only. Static mappings of specific MPI applications are evaluated in (MERCIER; CLET-ORTEGA, 2009; BRANDFASS; ALRUTZ; GERHOLD, 2012; ITO; GOTO; ONO, 2013; RODRIGUES et al., 2009) for the NAS-MPI applications, a Computational Fluid Dynamics (CFD)-kernel, a weather forecast application, and an application based on domain decomposition, respectively. None of the proposals supports dynamic mapping, they need previous information about the applications' behavior.

A related type of proposal that affects communication is based on what is called *communication reduction* or *communication avoidance* (BALLARD et al., 2014). Such proposals focus on reducing the impact of communication by reducing the amount of data that needs to be communicated between tasks in order to overcome the high performance and energy consumption impact of communication (SHALF; DOSANJH; MORRISON, 2010). These proposals focus on improving parallel algorithms, and can be seen as orthogonal to our techniques, since even a reduced amount of communication can be optimized with a better task mapping. A comprehensive overview of communication avoidance algorithms is given in Ballard et al. (2014).

6.2.2 Multithreaded Applications

In most multithreaded applications, communication is performed implicitly through memory accesses to shared memory areas. In (CRUZ et al., 2011; DIENER et al., 2010;

DIENER, 2010; BARROW-WILLIAMS; FENSCH; MOORE, 2009; BIENIA; KUMAR; LI, 2008), techniques to statically collect the communication pattern of the threads of parallel applications based on shared memory were evaluated. These methods consist of instrumenting simulators to generate memory accesses traces, which are analyzed to determine the communication pattern of the applications. The application is then executed with static mappings based on the detected patterns. Due to the high overhead of simulation, these mechanisms were only evaluated with small applications.

The ForestGOMP runtime system (BROQUEDIS et al., 2010a) uses information about the affinity between threads in OpenMP-based programs gathered from several sources, such as the developer, application source code (from OpenMP parallel sections), and hardware counters during runtime. Threads with a high affinity are then placed close to each other in the processor hierarchy. Due to its limitation to OpenMP applications with a custom runtime, ForestGOMP is not a general solution for task mapping.

The usage of the instructions per cycle (IPC) metric to guide thread mapping is evaluated in Autopin (KLUG et al., 2008). Autopin itself does not detect communication, but measures the IPC of several mappings fed to it and executes the application with the thread mapping that presented the highest IPC. In (RADOJKOVIĆ et al., 2013), the authors propose BlackBox, a scheduler that selects the best mapping by measuring the performance that each mapping obtained, similar to Autopin. When the number of threads is low, all possible thread mappings are evaluated. When the number of threads makes it unfeasible to evaluate all possibilities, the authors execute the application with 1000 random mappings to select the fastest one.

Several mechanisms use indirect communication statistics from hardware counters to perform the task mapping during execution. Azimi et al. (2009) use hardware counters that provide memory addresses of requests resolved by remote cache memories. It detects incomplete communication patterns, since memory requests resolved by local caches are not considered. Moreover, these hardware counters are specific to each processor architecture, and need to be adapted for each new architecture.

In (CRUZ; DIENER; NAVAUX, 2012; CRUZ; DIENER; NAVAUX, 2015), the communication pattern is detected by comparing the contents of the Translation Lookaside Buffer (TLB). The most recently used pages of a core have a corresponding entry in its TLB. The mechanism compares the contents of all TLBs in the system, and consider as communication when the same entry is found in the TLBs of different cores. Most current hardware architectures require hardware changes to support, since the TLB contents are usually managed by the processor and can not be accessed by software. In (CRUZ et al., 2014a), cache line invalidation messages are monitored to determine which cores access the same cache line. Each invalidation message is considered as a communication event, while the aggregated amount of communication of all cores is used to estimate the global behavior. This mechanism requires extensive hardware changes since it adds a vector to each core to store the number of invalidation messages received by all other cores.

Table 6.1: Summary of the main related task mapping mechanisms.

Mechanism	Type	Metric	Message passing	Shared memory	Multiple applications	Balance	Locality	No hardware change	Hardware independent	No application change	No previous execution
MPIPP	Semi.	Messages	✓		✓		✓	✓	✓	✓	
Autopin	Auto.	IPC		✓	✓		✓	✓	✓	✓	✓
Azimi et al.	Auto.	Cache misses		✓	✓		✓	✓	✓	✓	✓
ForestGOMP	Auto.	Shared data		✓			✓	✓	✓		✓
CDSM	Auto.	Page faults	✓	✓	✓	✓	✓	✓	✓	✓	✓

6.2.3 Summary of Related Work on Task Mapping

Compared to previous work, our proposal, CDSM, presents several advantages. It operates completely during execution and requires no previous knowledge of the application behavior. Furthermore, CDSM requires no changes to the hardware, applications or their runtime libraries. Since it works on the operating system level, all parallel applications that use memory accesses to communicate are supported. Table 6.1 contains an overview of the main related work in communication-based task mapping. We note that none of the previous proposals discuss a balance-based policy, focusing only on improving locality. We will compare CDSM to Autopin and MPIPP in Section 6.5.5.

6.3 CDSM: Communication Detection in Shared Memory

This section describes how CDSM detects the communication of parallel applications. We begin by explaining the general concept of the detection, followed by a description of the implementation in the Linux kernel and an analysis of the detection overhead.

The only requirement of CDSM is that the hardware and OS use virtual memory with paging. When an application causes a page fault, the hardware notifies the OS, which updates the page table of the process that caused the fault. It is important to note that threads of the same process share a common page table in many OS, such as Linux. Different processes have separate page tables. Processes and threads are handled similarly by CDSM. To simplify the explanation, we refer to both of them as tasks in this chapter and will only refer to processes and threads when they are handled differently.

6.3.1 Fundamentals of CDSM

The communication detection of CDSM is based on two fundamental concepts, which will be explained in this section. An overview of the operation of CDSM is shown in Figure 6.1.

6.3.1.1 Analyzing Memory Access Behavior Through Page Faults

Memory accesses are usually handled directly by the hardware, without any knowledge of the OS. However, whenever a memory address is accessed and its corresponding page is not set as present in the page table, the OS is notified of the faulting address. We make use of these page faults for the communication detection. The mechanism works as follows.

Whenever a page fault is caused by the parallel application, CDSM stores the ID of the task that generated the page fault in a hash table that is indexed by the physical memory address that caused the fault. It is necessary to use the physical address instead of the virtual address since different processes occupy different virtual address spaces. Since the full address is available during the page fault, communication can be detected using different block sizes. The memory is separated into *memory blocks* of an adjustable block size. In this way, the communication detection is independent of the page size that the hardware uses.

When another task causes a page fault in the same memory block, we add the task ID to the hash table and consider this fault a *communication event*. The amount of communication events of the parallel application is stored in a *communication matrix*. Each cell (x, y) of the matrix contains the number of communication events between processes x and y.

6.3.1.2 Increasing Detection Accuracy by Enabling Multiple Page Faults per Page

An important aspect of the detection mechanism is that, under normal circumstances, each process causes only one page fault per page, at the first access to the page. To increase the accuracy of the detection and to detect changes in the communication behavior during the execution, CDSM enables extra page faults, such that more than one fault can happen in the same page. CDSM periodically iterates over the page tables of the parallel application and modifies their entries.

The page table entries can be modified in different ways. A generic solution is to clear the page present bit, which is available in most architectures that support paging. In some architectures, such as x86_64, it is possible to modify reserved bits in the page table entry, which leads to a page fault that is slightly easier to resolve. In any case, it is necessary to remove the page table entry from the Translation Lookaside Buffer (TLB) after it is modified. As the extra page faults do not represent missing information in the page table, they can be resolved quickly by CDSM itself, without intervention from the normal kernel routines. In this way, the overhead of these extra page faults is minimized.

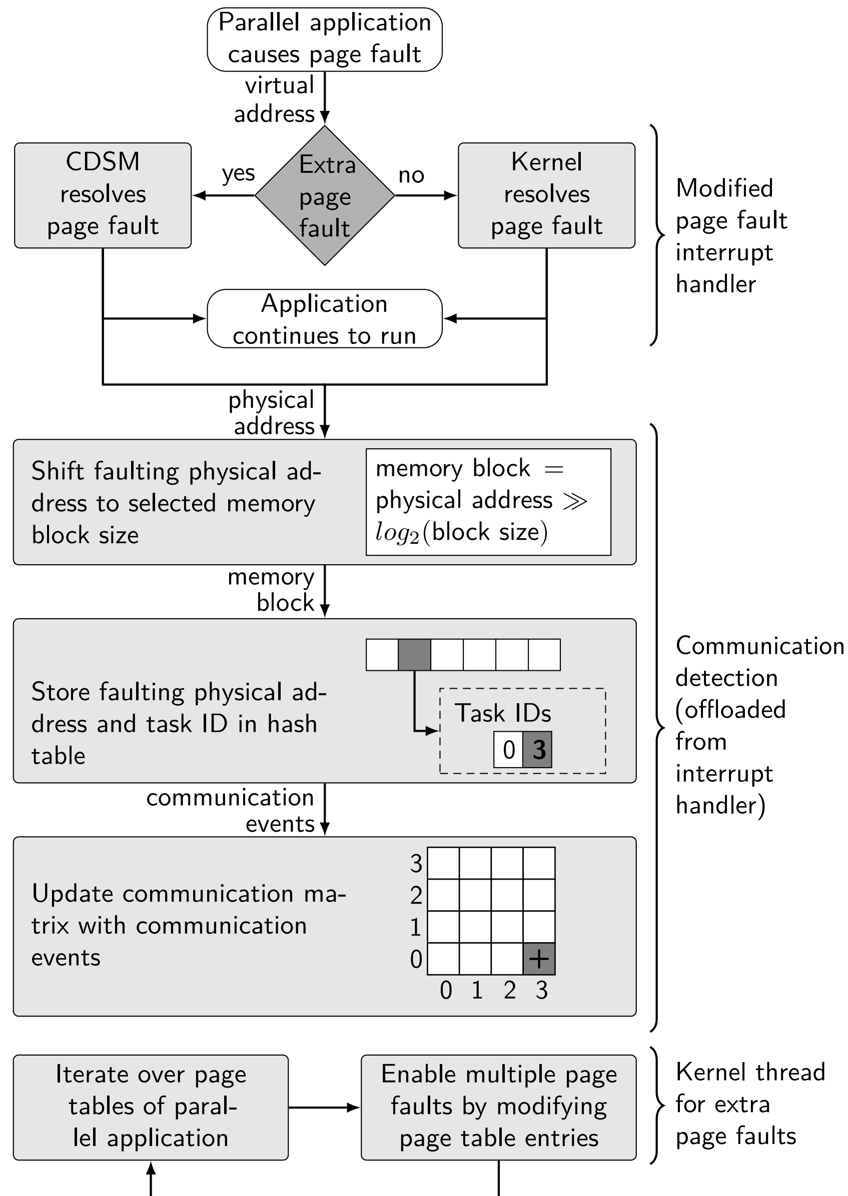


Figure 6.1: The communication detection mechanism.

6.3.1.3 Example of the Operation of CDSM

Figure 6.2 shows an example of the operation of CDSM for a parallel application consisting of 4 tasks. On application start, CDSM allocates data structures for the new application (a hash table and a communication matrix). After the application starts to execute, task 0 tries to access a memory address in a page that was never accessed before and generates a page fault. CDSM calculates the memory block of the address by bit shifting the address with the number of bits that correspond to the block size, and adds task 0 to the list of tasks that accessed this block. As the block has not been accessed before, the page fault does not represent a communication event and it is resolved by the kernel.

During the execution, CDSM enables extra page faults for the same page in the page table of task 3. The next time task 3 tries to access the page, it will therefore

generate an extra page fault. As the memory block has been accessed before, CDSM will count this fault as a communication event and increment the communication matrix in cell $(0, 3)$, which corresponds to task IDs 0 and 3. This situation is also depicted in Figure 6.1. CDSM is responsible for this extra page fault and resolves it by removing the modification to the page table. It then returns directly to the application, without calling the page fault handler of the OS.

6.3.2 Implementation

For each parallel application, two data structures need to be allocated. A hash table, where CDSM stores information about the page faults and the task IDs that caused them, and a communication matrix to store the number of communication events between tasks. The size of the hash table scales linearly with the size of the main memory. For each GByte of main memory, it consumes about 2.3 MByte of memory for the configuration used in our experiments. We allocate a communication matrix that is sufficient to store the number of communication events for up to 1024 tasks. Each matrix cell is 4 Bytes wide, for a total memory consumption of 4 MByte per parallel application.

CDSM was implemented as a Linux kernel module for the x86_64 architecture. The implementation consists of three parts: a modification to the page fault interrupt handler, the communication detection, which is offloaded from the interrupt handler, and a kernel thread that enables extra page faults. The description follows the sequence shown in Figure 6.1.

The default page fault behavior of the kernel was changed by adding a kprobe (MAVINAKAYANAHALLI et al., 2006) to the page fault interrupt handler. When a parallel application causes a page fault, CDSM determines in the modified handler if it was responsible for the page fault, and resolves the fault itself or lets the normal kernel routines handle it. CDSM then schedules a work item in the kernel work queue. The item consists

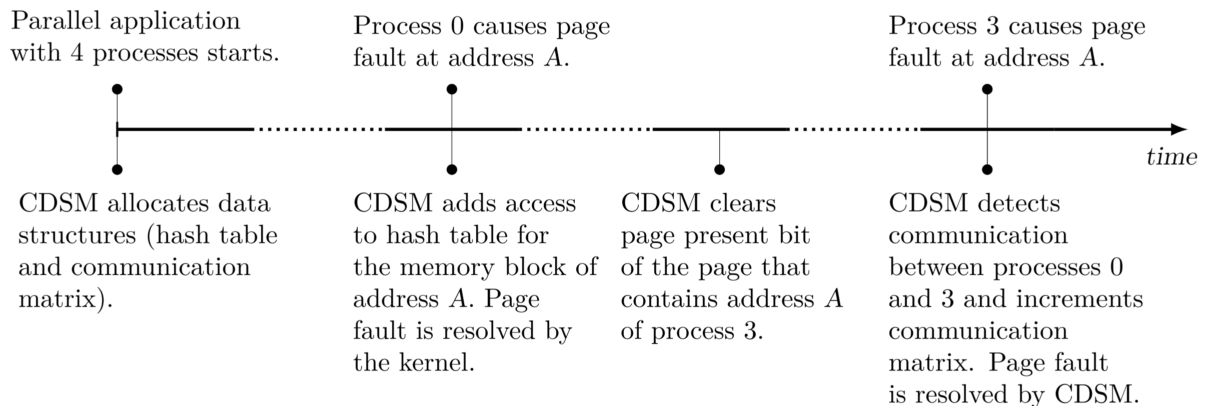


Figure 6.2: Example of the operation of CDSM for a parallel application consisting of multiple tasks.

of the task ID that caused the fault and the physical address of the fault. Afterwards, the parallel application continues to run.

The work item is analyzed by the communication detection routine. In the routine, the physical address is bit shifted to the chosen memory block size. With the shifted address, CDSM accesses the hash table and updates the list of task IDs that accessed the memory block. If the memory block has been accessed before by different tasks, CDSM increments the number of communication events in the communication matrix for each pair of tasks that accessed the memory block. We use the relaxed communication detection mechanism presented in Section 3.1.5, and use the same default configuration with 2 task IDs per memory block, but on a higher granularity.

The handling of the page faults and the communication detection were separated for two reasons. First, it allows the parallel application to continue executing earlier, before detecting the communication. Second, it allows the communication detection to be performed during a non-critical time, such as when the application is stalled due to a cache miss.

To enable multiple faults per page during the execution of the parallel application, we create a kernel thread that periodically iterates over the page tables of the parallel application and enables extra page faults. To increase the accuracy of the detection, page faults are only enabled in virtual memory areas that can be used for communication. For processes, memory areas that can be used for communication are marked as *shared* by the virtual memory subsystem of Linux, and the kernel thread only modifies the page table for these shared areas. For threads, which share the same virtual address space, communication can happen in more areas, so the kernel thread enables extra page faults in a larger part of the virtual memory space. In both cases, extra page faults are only enabled for data segments, not for code segments.

After identifying a suitable virtual memory area, the kernel thread enables the faults by setting a reserved bit in the page table entry and clears the corresponding entry in the TLB. In this way, extra page faults can be detected and resolved already in the interrupt handler. To limit the overhead of the mechanism, the kernel thread enables extra page faults up to a percentage of the number of pages the application accesses per second. We selected a limit of 1% of the pages the application uses per second. This value was chosen to provide a high level of accuracy with a negligible overhead, as will be shown in Section 6.5, and will compare it to other limits.

6.3.3 Calculate the Task Mapping

The task mapping problem is defined as finding a mapping of tasks to processing units (PUs) that maximizes the locality, given a description of the communication behavior and the hardware hierarchy. Several algorithms have been suggested previously to calculate this mapping. We selected the EagerMap mapping algorithm for CDSM, which was described in Section 3.3, and perform the experiments with it.

We ported EagerMap to the kernel as part of CDSM. The hardware topology (including information about caches and NUMA nodes) is generated directly from the

information available in the kernel, without the need for external tools. During the execution of the parallel application, CDSM periodically fetches the sharing matrix and applies the EagerMap algorithm on the matrix and the hardware topology. The output of EagerMap is the PU where each task should execute. CDSM then migrates the tasks to their assigned PUs.

We selected a mapping interval of 200 ms, below which we did not measure further gains from more migrations. Due to the high stability of EagerMap compared to other mapping algorithms, unnecessary task migrations due to small changes in the communication matrix are reduced (CRUZ et al., 2015).

6.3.4 Runtime Overhead of CDSM

To perform the communication detection, CDSM introduces two types of overhead to the execution of the application. The first type of overhead consists of the extra page faults, which require additional context switches to the OS as well as periodic accesses to the page tables of the parallel application to enable the extra page faults. This overhead is reduced by keeping the amount of computation in the page fault handler low. The second type of overhead is the communication detection itself. It consists of accesses to the hash table and the communication matrix, which have a constant time complexity. Apart from the runtime overhead, CDSM also requires memory to store the hash table and communication matrix, as discussed in Section 6.3.2. In Section 6.5, we will evaluate the runtime overhead of CDSM quantitatively.

6.3.5 Summary

The communication detection part of CDSM has the following features. CDSM is independent of the parallelization model, supporting any model that can use shared memory to communicate, as well as applications that use several models. CDSM detects communication online, during the execution of the parallel application, and requires no modification to the application, its runtime system or expensive operations such as tracing. CDSM is also highly OS and hardware independent. It can be applied to all platforms that use virtual memory with paging, which covers most of the current computer architectures. No architecture specific information, such as a special hardware counter, is necessary.

6.4 Methodology of the Experiments

For the evaluation of task mapping in this section, we use our main evaluation system, the *Xeon* machine system described in Section 2.2. We use the MPICH2 (GROPP, 2002) MPI framework, which uses Nemesis (BUNTINAS; MERCIER; GROPP, 2006b) for shared memory communication by default. We also verified the correct operation of CDSM with Open MPI (GABRIEL et al., 2004). Table 6.2 summarizes the configuration of CDSM used in the experiments.

Table 6.2: Experimental configuration of CDSM.

Parameter	Value
Detection	256 Bytes memory block size
Hash table	4 Million elements, 73 MByte memory usage
Page faults	max. 1% of pages/second extra
Mapping interval	200 ms

For the experiments, we use the parallel applications presented in Section 2.1: NAS-OMP, NAS-MPI, PARSEC, HPCC, NAS-MZ, as well as the BRAMS weather prediction model. We execute most benchmarks with 64 threads or processes, as *Xeon* can execute 64 tasks simultaneously. The NAS-MZ benchmarks were executed with 4 processes and 16 threads per process. All NAS benchmarks were executed with the *B* input size. All other benchmarks were executed with the input parameters specified in Section 2.1.

6.5 Evaluation of CDSM

In this section, we present two main sets of experimental results: the accuracy of the communication patterns detected by CDSM, as well as performance and energy consumption improvements. Furthermore, we analyze the impact of different numbers of page faults on the gains achieved by CDSM, as well as the overhead.

6.5.1 Accuracy of Detected Communication Behavior

An important aspect of CDSM is its communication detection accuracy. Compared to the relaxed definition of communication introduced in Section 3.1.5, CDSM has two sources of inaccuracy. First, it uses a hash table with a small fixed size to determine the communication, as opposed to a table with an infinite size that covers the full memory range of the application. Second, by using a sampling mechanism that only uses a small percentage of memory accesses as opposed to every memory access. This section evaluates the accuracy of CDSM with the default configuration presented Section 6.4. The accuracy and performance achieved by varying the number of extra page faults will be evaluated in Section 6.5.4.

6.5.1.1 Methodology of the Comparison

We verify the accuracy of CDSM by comparing the generated communication matrices of CDSM with tracing-based communication detection using the Mean Squared Error (MSE) introduced in Section 3.1.4. For the benchmarks that are based only MPI (NAS-MPI and HPCC), we use the eztrace tool (TRAHAY et al., 2011), since it offers a perfectly accurate analysis of the communication behavior by instrumenting all MPI functions that exchange data. For the other benchmarks that also use implicit communication, we use our numalize tool presented in Chapter 5.

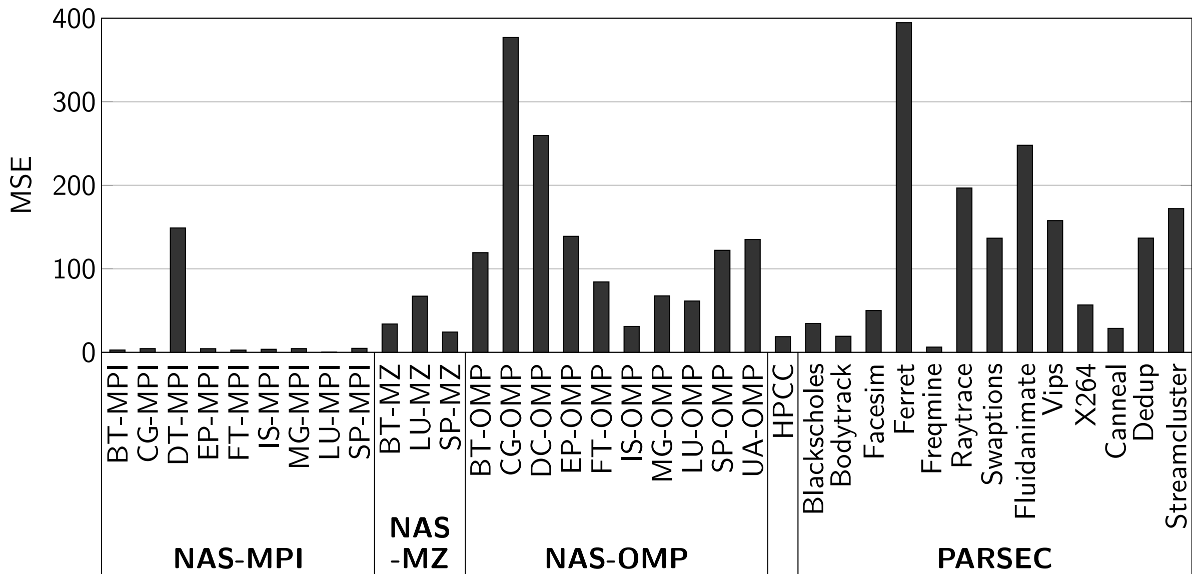


Figure 6.3: Communication detection accuracy of CDSM compared to eztrace (NAS-MPI and HPCC) and numalize (all others). The figure shows the Mean Squared Error (MSE) between the normalized communication matrices. Lower values are better.

We performed the accuracy experiments with 64 tasks on the *Xeon* machine. We begin with an evaluation of the global application behavior, comparing the communication patterns that were detected during the whole execution. We also analyze the detection speed of CDSM, showing how long it takes to reach a certain communication pattern. For the NAS benchmarks, we used the *A* input size to limit the time during which CDSM can detect the behavior. The HPCC and PARSEC suites were executed with their default configuration.

6.5.1.2 Global Behavior

Figure 6.3 shows the MSE of the communication detected by CDSM and by the eztrace/numalize tools. Lower values indicate a higher accuracy. The theoretical maximum MSE for this configuration is 9844. All NAS-MPI benchmarks except DT-MPI and HPCC have an MSE of almost zero, indicating a very high accuracy. Since these benchmarks use relatively small memory areas to communicate, it is easy for CDSM to detect this communication. For the DT-MPI benchmark, the MSE is higher since its communication behavior changes between executions.

For the benchmarks that use implicit communication, the MSE is generally higher, though it remains below 200 for most benchmarks. The applications with the highest MSEs (CG-OMP, DC-OMP and Ferret) have very unstructured patterns, which increases their susceptibility that small changes in the detected communication increase the MSE. For all applications, the MSE is below 400, which is a comparable error with a granularity between 1 KByte and 16 KByte, as discussed in Section 3.1.5.2. These results indicate that CDSM's accuracy with the presented configuration is sufficient for an optimized task mapping.

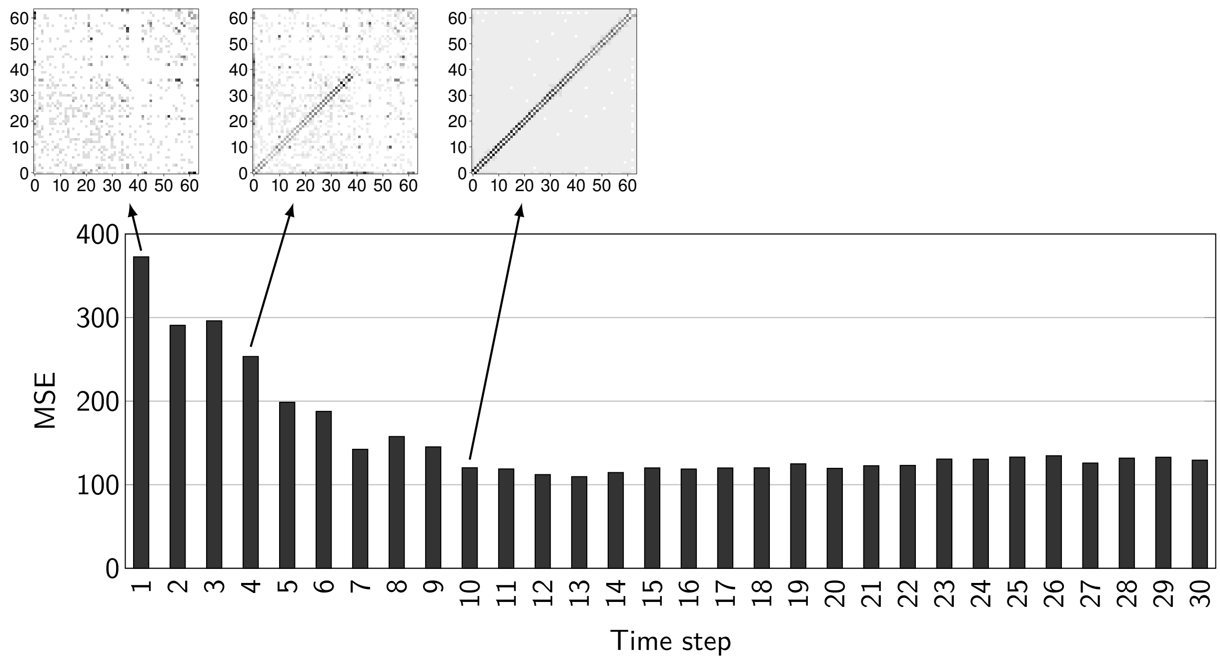


Figure 6.4: Dynamic detection behavior of the SP-OMP benchmark. 30 out of 200 time steps are shown.

6.5.1.3 Dynamic Behavior

To measure the time it takes CDSM to detect a communication pattern, we execute the SP-OMP benchmark with the A input size and track the change of the communication matrix over time. At each 0.1 s of the execution (corresponding to about 1 time step out of 200), CDSM writes the detected communication pattern to a file. We then calculate the MSE for each of the patterns compared to the matrix generated with numalize.

Figure 6.4 shows the results of this experiment for the first 30 time steps, including the matrices for the first, fourth and tenth step. After the first step, there is no visible pattern, with homogeneous communication. On the fourth step, the pattern begins to develop and the correct behavior with a nearest-neighbor communication pattern is already visible. After the tenth step, the behavior is already very clear, and it changes only slightly during the rest of the execution. This means that, with only 0.5 sec of communication detection, CDSM was able to detect the communication behavior correctly, after about 2.5% of the application was executed.

6.5.2 Performance Results

To evaluate the performance improvements of CDSM, we measure performance improvements as well as reduction of the number of L3 cache misses with the Intel PCM tool (INTEL, 2012a), normalized to the OS results. We experimented with CDSM, comparing it to three other task mapping mechanisms, the default mapping of the operating system, a compact mapping and an oracle mapping. These mechanisms are described

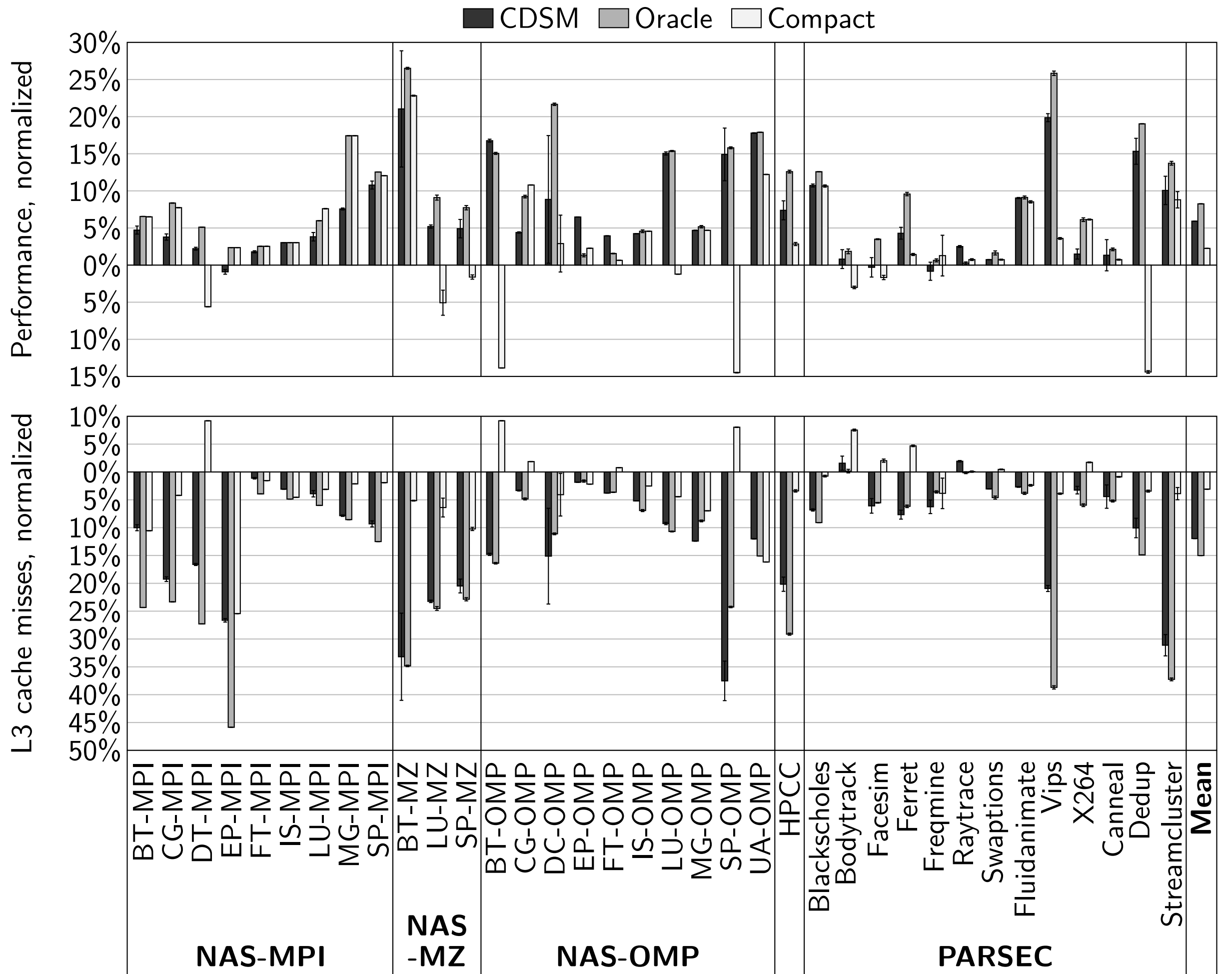


Figure 6.5: Performance results of CDSM on the *Xeon* machine, normalized to the OS.

in detail in Section 3.3, and we give only a brief overview here. The *Operating System (OS)* mapping uses the default scheduler (CFS) of the Linux kernel version 3.8. It represents the baseline for our experiments. In the *Compact* mapping, neighboring tasks are mapped sequentially to nearby cores in the topology, similar to the compact mapping that is available in some MPI and OpenMP frameworks (INTEL, 2012b; ARGONNE NATIONAL LABORATORY, 2014b). This mapping is a simple mechanism to optimize communication performance when most of the communication happens between directly neighboring processes. The *Oracle* mapping represents a perfect communication-aware mapping mechanism. We generate it by collecting communication traces and performing an optimized static mapping of tasks to cores with the mechanism described in Chapter 5. This mapping mechanism optimizes the execution in terms of the communication between the processes and has no runtime overhead. *CDSM* was executed with the default configuration presented in Section 6.3.

Figure 6.5 shows the results of the task mapping in terms of performance improvements and reduction of L3 cache misses. The graphics are normalized to the result of the Linux OS, our baseline. From the NAS-MPI benchmarks, we classified BT-MPI, LU-MPI, MG-MPI and SP-MPI as having a large amount of communication between the processes.

These applications show large improvements of execution time, reaching up to 10.8% with CDSM for SP-MPI. DT-MPI shows relatively small improvements due to its very low execution time. CG-MPI with its cluster communication pattern showed moderate improvements. The EP-MPI and FT-MPI benchmarks, which we classified as having small amounts of communication, only show small differences in execution time with all mapping mechanisms. Likewise, IS-MPI shows only small improvements because the amounts of communication between the processes are very similar. Due to their relatively simple and static communication structure, most NAS-MPI benchmarks also benefit from the Compact mapping.

The NAS Multi-Zone benchmarks show similar improvements of the performance, with up to 21.0% for BT-MZ with CDSM. LU-MZ and SP-MZ show moderate performance improvements with similar values for CDSM and the Oracle. On the other hand, the Compact mapping reduces performance for both benchmarks, by 5.2% and 1.6% respectively.

Several NAS-OMP benchmarks, such as BT-OMP, SP-OMP, and UA-OMP, have large amounts of communication between the directly neighboring threads. In these cases, the Oracle and CDSM produce similar performance improvements (up to 14.9% for SP-OMP with CDSM). However, as BT-OMP and SP-OMP are imbalanced, the Compact mapping reduces performance substantially, by about 14%. In the benchmarks where the communication is not performed by direct neighbors, but by threads that are further away (DC-OMP, LU-OMP, and MG-OMP), CDSM achieves much higher speedups than the Compact mapping. EP-OMP and FT-OMP have a very low amount of communication and therefore present only small improvements. IS-OMP can not benefit from an improved mapping due its all-to-all communication behavior, and shows only small improvements as well.

HPCC shows medium improvements, reducing execution time by 7.4% with CDSM. Due to its complex and changing communication behavior, the compact mapping only presents very small improvements.

From the PARSEC benchmarks, Blackscholes shows moderate improvements of 10.7% with CDSM. Considering only the parallel phase of the application, the speedup is much higher, since it has a very long sequential phase (SOUTHERN; RENAU, 2015). Raytrace, X264 and Canneal show only small improvements, due to their small amounts of communication and homogeneous patterns. Fluidanimate with its highly structured communication behavior shows moderate improvements despite the relatively low amount of communication. Vips showed the highest improvements of all PARSEC benchmarks, of 19.9% with CDSM. The pipeline communication patterns of Ferret, Dedup and Streamcluster show also large improvements. Facesim, Freqmine and Swaptions show relatively unstructured communication patterns and have moderate performance improvements.

It is important to note that in all cases, the improvements achieved by CDSM were very close to the improvements of the Oracle mapping. Furthermore, even in applications whose communication behavior can not be exploited for mapping, the performance is not reduced. The geometric mean of all performance improvements was 5.9% using CDSM and by 8.3% using the Oracle. The Compact mapping only improved the performance slightly compared to the OS, with a geometric mean of 2.3%, since only benchmarks

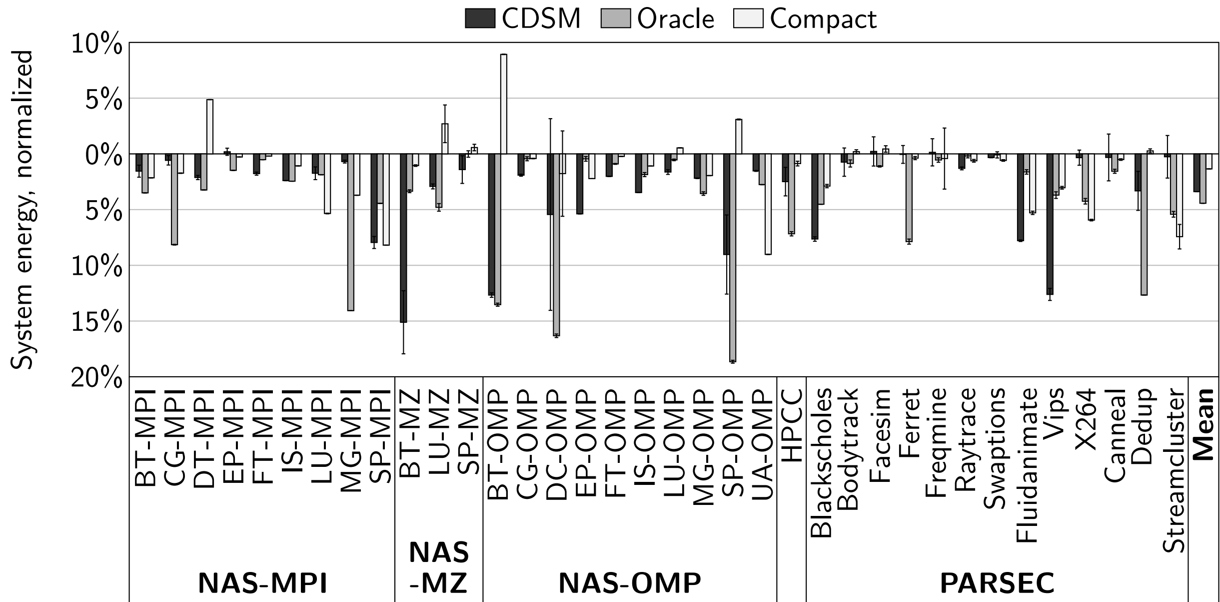


Figure 6.6: Energy consumption results of CDSM, normalized to the OS.

with a clear communication pattern between neighboring tasks (such as BT-OMP and SP-OMP) can benefit from it. Most benchmarks show more complex patterns which limit the improvements from the Compact policy. Moreover, it actually reduces performance for some benchmarks.

The reason for the performance improvements can be seen in the reduction of L3 cache misses in Figure 6.5. On average, CDSM reduced the number of L3 misses by 12.0%, compared to 15.1% for the Oracle and 3.1% for the compact mapping. Overall improvements were slightly higher than the execution time. This is expected, as a communication-aware mapping directly improves cache usage but has only an indirect influence on the execution time, through fewer cache misses and improved usage of the interconnections. Most of the benchmarks show improvements that are qualitatively similar to their performance improvements.

6.5.3 Energy Consumption

Reducing the execution time will usually lead to a reduction of the energy consumption in most cases. Communication-aware task mapping also causes a more efficient usage of the processor resources, as discussed in Section 5.1.2.2, which can lead to additional energy savings. We measure the energy consumption of the *Xeon* machine during the execution of each application with the help of the Baseboard Management Controller (BMC), which exposes the energy consumption of the system through Intelligent Platform Management Interface (IPMI). We experiment with the same mapping mechanisms and configurations as in the performance evaluation.

Figure 6.6 shows the results for the *Xeon* system energy consumption. For all benchmarks, CDSM reduces the energy consumption. Similar to the performance results, we

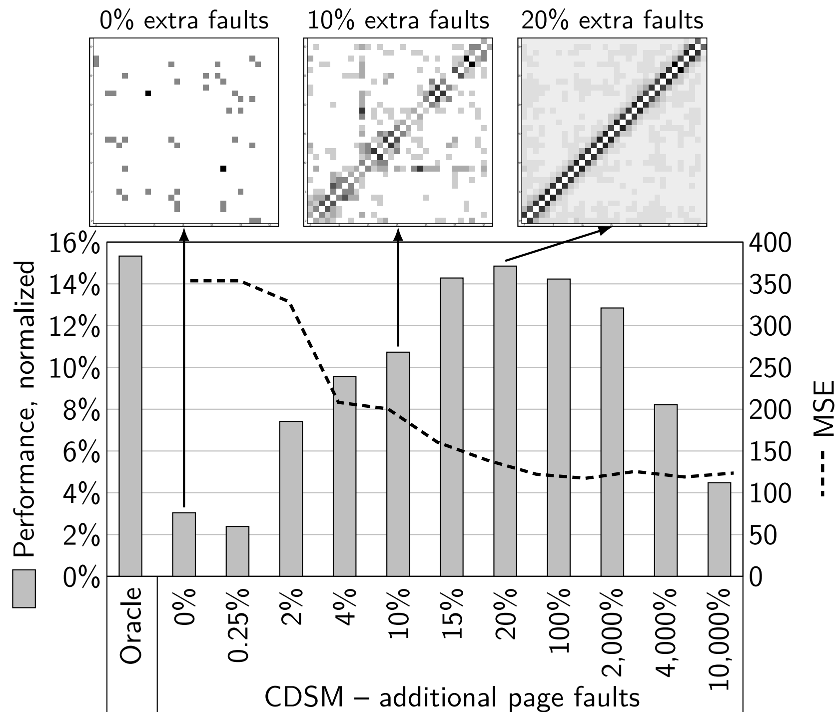


Figure 6.7: Varying the number of extra page faults of the SP-OMP benchmark (with the A input size). Three communication matrices for different numbers of additional page faults are shown above the performance results.

save more energy for benchmarks that have more communication with a more structured pattern. Energy consumption was reduced most in the BT-MZ benchmark, with a reduction of 15.1% with CDSM. On average, energy consumption was reduced by 3.4% with CDSM and 4.5% with the Oracle. As before, the Compact mapping only reduced energy consumption slightly (1.3% on average).

6.5.4 Varying the Number of Page Faults

The improvements of CDSM are sensitive to the number of extra page faults. Increasing the number of extra page faults increases the detection accuracy, but also increases the overhead on the running application. This section evaluates this tradeoff.

6.5.4.1 Methodology

We selected the SP-OMP benchmark from NAS-OMP with the A input size, since it has the most illustrative behavior. We execute CDSM with various numbers of additional page faults, comparing the results to the Oracle mapping and the OS.

6.5.4.2 Results

The performance results of this experiment are shown in Figure 6.7. Values are normalized to the OS. On the horizontal axis, we show the improvements of the Oracle and the improvements of CDSM with various numbers of extra page faults, given as a

percentage of the original number of faults. The figure also shows the sharing matrices for three different percentages of extra faults, 0%, 10%, and 20%. Even for 0% of extra faults, CDSM improves performance slightly compared to the OS, despite the inconclusive sharing matrix. The reason is that CDSM is able to reduce the number of unnecessary thread migrations compared to the OS.

When increasing the number of extra faults, the gains from CDSM are rising as well. At 10% of extra faults, the communication pattern already becomes visible, with large amounts of communication between neighboring threads. At 20% of extra faults, CDSM reaches its maximum improvements, which are very close to the Oracle. Increasing the faults further does not improve the performance gains, and the overhead begins to affect the application. However, even with 10,000% of extra faults (corresponding to extra 100 faults per page), performance is still higher than the OS. This experiment shows that even with very little information about the application behavior (less than 1 extra fault per page), task mapping can be performed effectively.

6.5.5 Comparing CDSM to Related Work

We compared CDSM to two of the state-of-the-art mechanisms that were described in Section 6.2. For the MPI-based applications (NAS-MPI and HPCC), we used the MPIPP toolkit (CHEN et al., 2006). MPIPP uses MPI communication traces to perform a static mapping of MPI ranks to cores, and is therefore similar to our Oracle mapping for the MPI-based benchmarks.

For the multithreaded applications (NAS-OMP and PARSEC), we compared CDSM to Autopin (KLUG et al., 2008). Autopin periodically migrates the threads of the parallel application using a set of previously generated mappings. For each application, we executed Autopin with 5 different mappings, consisting of the mapping used by our Oracle mechanism, the compact mapping and 3 random mappings. After an initialization phase of 500 ms, each of the 5 mappings is executed for 200 ms. The mapping with the highest IPC over all threads is then maintained for the next 10 seconds, after which we evaluate the 5 mappings again. As we are not aware of a previous solution that handles hybrid applications, we do not use the NAS-MZ applications in the comparison.

Figure 6.8 shows the performance improvements of the benchmarks. As expected, MPIPP shows almost the same improvements as CDSM for the NAS-MPI benchmarks. However, it is not able to fully take advantage of the dynamic communication behavior of HPCC and achieves results that are not optimal. For the NAS-OMP benchmarks, Autopin produces reasonably good results, due to their static communication behavior. For PARSEC, Autopin results are much worse and actually increases execution time for several benchmarks. Overall, the two related mechanisms achieved improvements of 4.9% on average, compared to 11.1% for CDSM.

6.5.6 Case Study: BRAMS

To show the operation of CDSM on a real-world scientific application, we selected the BRAMS weather prediction model (FREITAS et al., 2009) introduced in Section 2.1, and executed it on the *Xeon* machine that was used in the previous experiments in this chapter. BRAMS is based on MPI, and we execute it with 64 processes and the light1gr input set. All evaluation parameters, including the configuration of CDSM, are the same as in the previous experiments.

6.5.6.1 Communication Behavior

The communication pattern of BRAMS consists of three phases, which are depicted in Figure 6.9. In the beginning of each time step, neighboring processes communicate for about 25% of the duration of the time step (Figure 6.9a). Until the end of each time step, more distant processes communicate (Figure 6.9b). At the very end of each time step, a brief period with an all-to-all pattern (Figure 6.9c). The global communication pattern, shown in Figure 6.9d, is very similar to the pattern of the LU-MPI benchmark presented in Section 3.4.1.3. Due to this dynamic behavior, we modify the application for the Oracle mapping, to perform a task migration whenever the pattern changes.

6.5.6.2 Performance Results

The performance results for the BRAMS application are shown in Figure 6.10, comparing the OS, Compact, and Oracle mappings to CDSM. On average, performance

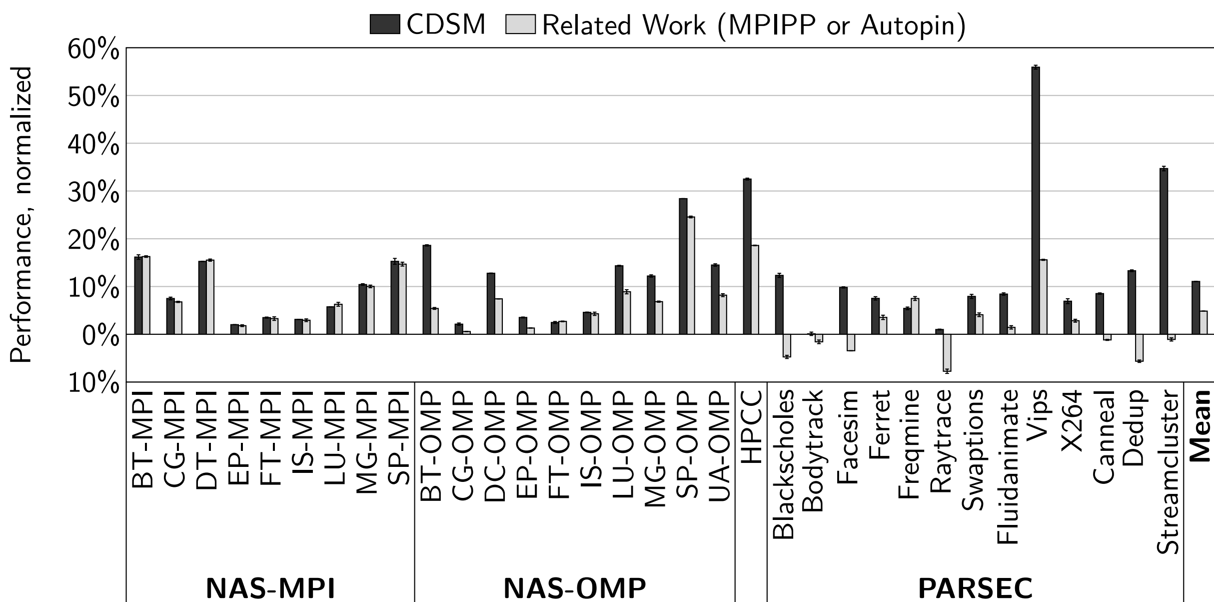


Figure 6.8: Application execution time when performing mapping with CDSM and related work. The values are normalized to the results of the OS. We use MPIPP (CHEN et al., 2006) (for NAS-MPI and HPCC) and Autopin (KLUG et al., 2008) (for NAS-OMP and PARSEC) as related work.

was improved by 10.9% with the Oracle and 8.6% with CDSM. Due to the dynamic behavior, the compact mapping only improves performance by 4.0%. The results show that CDSM not only can improve smaller benchmarks and application kernels, but also large real-world applications with longer execution times.

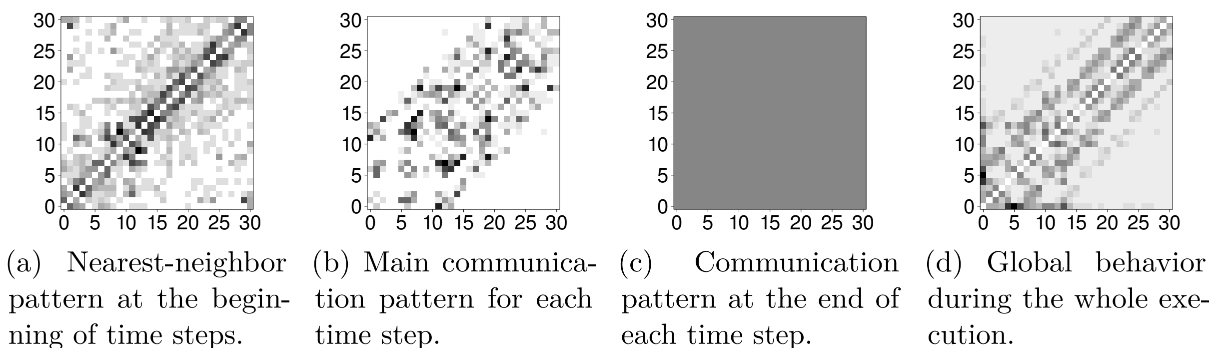
6.5.7 Overhead of CDSM

Because it operates during the runtime of an application, CDSM imposes an overhead on the execution. The overhead can be classified into three groups, the communication detection (including the extra page faults), the calculation of the mapping and the migration of the threads and processes. The communication detection consists of accesses to the hash table and accesses to the page tables of the application to enable multiple page faults per page. These extra page faults present an overhead for a small part of the memory accesses by the application. However, extra page faults were generated for less than 0.0001% of memory accesses of each application, corresponding to less than 25% extra faults for each application during execution.

The communication detection overhead was measured by statically mapping each application, and executing it only with the communication detection part of CDSM, without the mapping algorithm and the migration. The mapping overhead consists of the repeated execution of the mapping algorithm during the execution of the application. It was evaluated by measuring the time spent in the mapping function. The migration overhead consists of a small increase in cache misses for the process after the migration, as well as performing the task migration itself in the kernel via the `sched_setaffinity()` function.

The communication detection and mapping overheads are shown in Figure 6.11. The values are the percentage of execution time of each application. For all benchmarks, the communication detection and mapping overhead is less than 0.5%. The average overhead of the communication detection is 0.34% and 0.23% for the mapping, for a total average overhead of 0.57%.

For a more in-depth analysis of the overhead, we counted all events of CDSM that happened during execution of the FT-MPI benchmark, and measured the number of cycles spent for each event with the help of the Time Stamp Counter (TSC) of the processor.



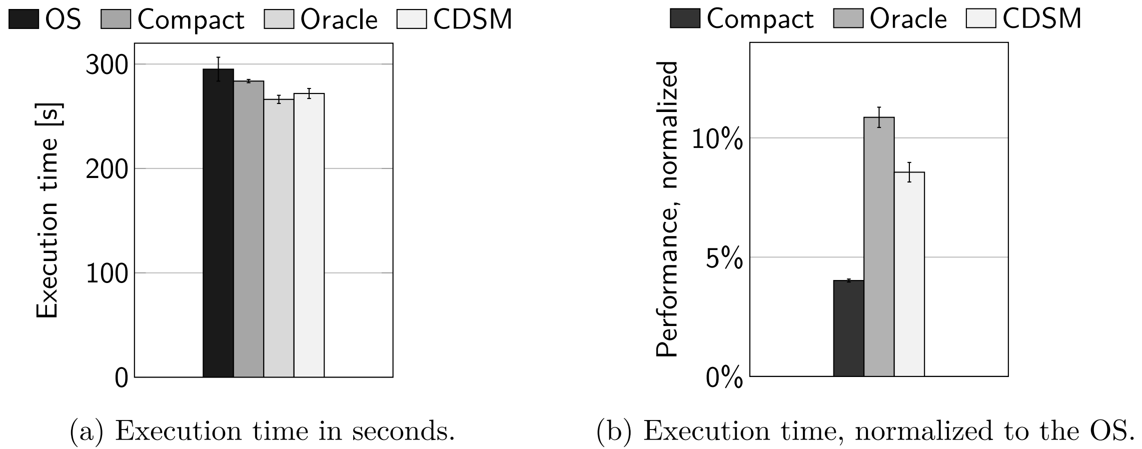
(a) Nearest-neighbor pattern at the beginning of time steps.

(b) Main communication pattern for each time step.

(c) Communication pattern at the end of each time step.

(d) Global behavior during the whole execution.

Figure 6.9: Communication behavior of the BRAMS application detected by CDSM.



(a) Execution time in seconds. (b) Execution time, normalized to the OS.

Figure 6.10: Performance results of BRAMS on *Xeon*.

FT-MPI was chosen since it had the highest overhead overall due to its large memory usage in a very short execution time. Table 6.3 shows the statistics for the normal page faults, extra page faults, data structure updates, calls to the mapping algorithm, and number of task migrations. We show the numbers of each event type, the time spent for each type, and the percentage of the total execution time of FT-MPI. We can see that the main overhead is generated by the extra page faults, update of data structures, and the execution of the mapping algorithm. The task migrations do not contribute significantly to the overall overhead.

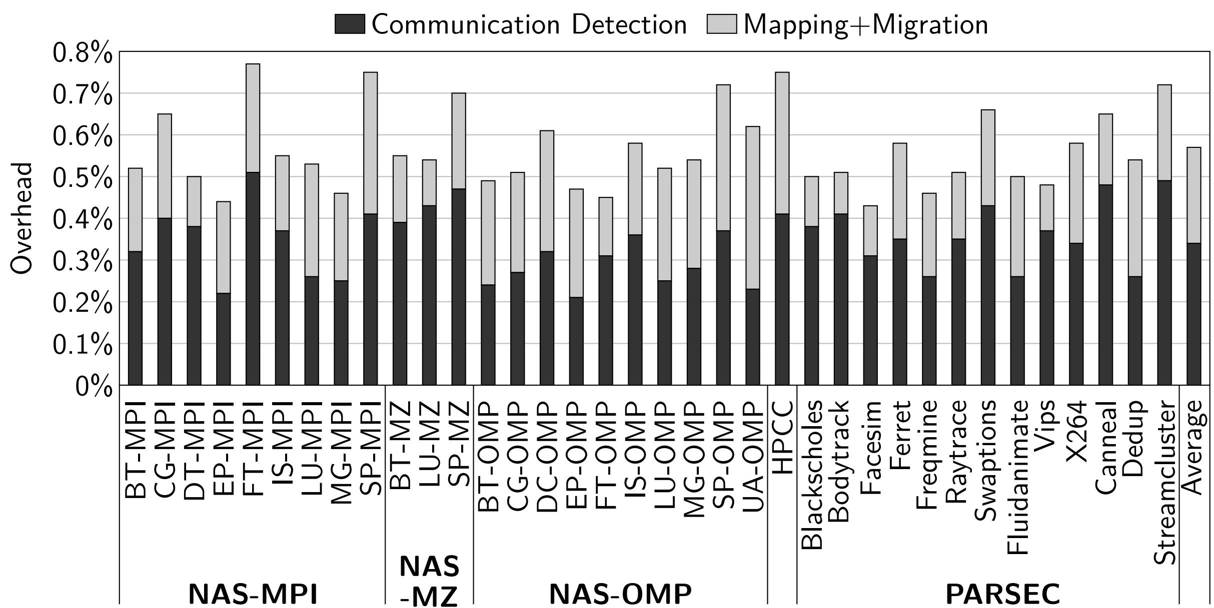


Figure 6.11: Overhead of CDSM in percent of the total execution time of each application.

Table 6.3: Overhead of the FT-MPI benchmark.

Value	Normal page faults	Extra page faults	Update data struct.	Calculate task mapping	Task migration
Number of events	247,817	51,236	299,053	573	34
Absolute time	173 ms	18.3 ms	12.3 ms	15.1 ms	3.2 ms
Overhead (% of total time)	—	0.31%	0.21%	0.26%	0.05%

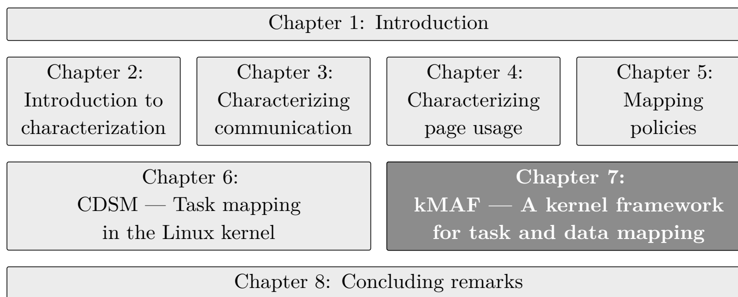
6.6 Summary

In shared memory architectures, mapping the tasks of parallel applications according to their communication behavior can reduce execution time and energy consumption. As communication is performed through memory accesses, it is necessary to efficiently analyze them. We introduced CDSM, a mechanism that performs communication-aware task mapping during the execution of the parallel application. It needs no modifications to the application or any previous knowledge about its behavior and works with a wide variety of parallelization paradigms that use shared memory to communicate.

Communication is detected by analyzing the page faults of parallel applications. CDSM enables extra page faults for the application to increase the accuracy of the detection. These page faults can be resolved quickly without involving the normal OS routines. The detected communication behavior is used by the mapping mechanism to calculate an optimized task mapping and migrate the tasks.

CDSM was evaluated with the MPI, hybrid MPI+OpenMP and OpenMP implementations of the NAS Parallel Benchmarks (NPB), the HPC Challenge (HPCC) benchmark, and the PARSEC benchmark suite. We verified that all communication patterns were detected correctly by comparing them to message and memory tracing methods. CDSM reduced execution time by up to 21.0% (5.9% on average) compared to the OS scheduler. System energy consumption was reduced by up to 15.1% (3.4% on average). Improvements were close to an Oracle mapping and significantly better than previously published mechanisms. Our results also showed that a simple compact mapping of processes and threads to cores can not achieve optimal performance for many applications. The overhead imposed by CDSM on the applications was less than 0.8%. Based on the ideas of CDSM that were presented here, the next chapter will introduce a combined thread and data mapping mechanism.

7 KMAF: INTEGRATED TASK AND DATA MAPPING



The second automatic mapping mechanism discussed in this thesis, the *kernel Memory Affinity Framework (kMAF)*, performs integrated task and data mapping on the kernel level. It reuses several of the ideas and concepts developed for CDSM and extends them for data mapping. kMAF will be presented and evaluated in this chapter. For the reasons discussed in Section 4.1.4, we will focus only on multithreaded applications in this chapter.

7.1 Introduction

On parallel architectures with a Non-Uniform Memory Access (NUMA) behavior, it is necessary to allocate each memory page on a specific NUMA node. Traditional allocation mechanism such as first-touch or interleave do not take the access pattern to the pages into account when deciding where to map pages. Our discussion in Section 4.4 showed that most parallel applications have a very high page access exclusivity, which suggests that memory access locality can be improved for most of them. However, the balance of memory access needs to be considered as well in order not to overload some NUMA nodes.

In Chapter 6, we showed that by analyzing the page faults of parallel applications, CDSM can accurately detect the communication pattern between the tasks and perform an optimized mapping of tasks to execution units. However, extending this mechanism to perform data mapping is not straightforward. Multiple challenges need to be resolved.

Increasing the number of extra page faults. To perform task mapping, it is only required to know how the tasks communicate. Therefore, it is possible to have an accurate communication pattern with a low number of extra page faults. On average, less than 1 extra page fault per page was introduced. For data mapping however, the information needs to be much more precise, because it is necessary to analyze the page usage of each page. Therefore, much more page faults must be generated (multiple faults per page) to perform the data mapping. This increases the overhead on the running application, both to generate the faults and evaluate the information gained from them.

Storing the information. Similarly, we need to store the information for data mapping for each page, and thus require an efficient way to store a large amount of data. For task mapping, it was sufficient to manage a communication matrix and a relatively small hash table to detect the communication. Furthermore, the information in the hash table had a small lifetime and can be overwritten in case of a conflict. The page usage has a much higher lifetime, because losing information would cause more

page migrations. For these reasons, the storage overhead for data mapping will be much higher.

Cost of page migration. Migrating pages involves copying data between nodes and has a higher overhead than migrating tasks. Therefore, care must taken not to overload the memory controllers with memory access operations and therefore hurting the performance of the system. This also includes preventing unnecessary ping-ponging of pages between nodes.

After a brief review of related work regarding data mapping, we will introduce kMAF, followed by an evaluation using parallel benchmarks.

7.2 Related Work on Data Mapping

In addition to the related work regarding thread and process mapping that was presented in Section 6.2, we present here the state-of-the-art in data mapping. We divide previous research on data mapping into several broad groups according to where information about the memory accesses behavior is collected: OS-based mechanisms, compiler or runtime library-based mechanisms, mechanisms that operate on the hardware level, and manual mapping.

7.2.1 OS-Based Mapping

Traditional data mapping strategies that have been employed by operating systems on NUMA architectures are *first-touch*, *interleave* and *next-touch*. In the first-touch policy (MARCHETTI et al., 1995), a page is allocated on the first NUMA node that performs an access to the page, and the page is never migrated. This is the default policy for most current operating systems, including Linux. In some circumstances, first-touch can lead to overloading of NUMA nodes, for example when a single thread initializes a large amount of data. In these cases, it can be beneficial to distribute memory pages more equally to balance the load on the memory controllers. The most common way to distribute pages is to use an interleave policy, which is available on Linux through the `numactl` tool (KLEEN, 2004). These simple policies do not take the actual memory access behavior of the parallel applications into account, which limits their applicability in modern hardware architectures. In next-touch policies, a page is marked in such a way that it will be migrated to the node that performs the next access to it. Löf et al. (LÖF; HOLMGREN, 2005) propose such a mechanism for the Solaris operating system. Several other proposals discuss similar techniques for the Linux kernel (GOGLIN; FURMENTO, 2009; LANKES; BIERBAUM; BEMMERL, 2010). In case a page is accessed by several nodes, such next-touch mechanisms can lead to excessive migrations.

Policies that improve on these basic mapping strategies on the OS level focus mostly on refining the data mapping during execution, as the OS has no prior information about the application behavior. Most mechanisms perform an online profiling of the application to guide mapping decisions. LaRowe et al. (1991, 1992) present an analytical model of page

placement for early NUMA architectures without hardware cache coherence. They propose page migration and replication policies, where the same page is stored on multiple NUMA nodes. Implemented in the DUnX research kernel, policy decisions are performed during page faults, with a page scanner daemon periodically triggering additional evaluations via extra page faults. The coherence between replicated pages is maintained with the same software mechanism that performs cache coherence. No balancing of pages or thread mapping is performed. Furthermore, replicating pages has a much higher complexity and overhead on modern cache-coherent NUMA (ccNUMA) machines, where cache coherence is maintained in hardware and page coherence needs to be performed in an additional software layer.

Verghese et al. (1996b, 1996a) propose similar dynamic page migration and replication mechanisms for SGI's IRIX operating system, but use cache misses as a metric to guide mapping decisions. They require information about all cache misses and migrate pages to a node with lots of cache misses from a single task or replicate a pages if it receives a lot of cache misses from multiple tasks. The authors also evaluate the number of TLB misses as a metric to guide the mapping, but conclude that it is not accurate enough for data mapping. For modern architectures, this detailed information about cache misses can not be gathered with an acceptable overhead. Cache misses themselves are a more indirect measure of the memory access behavior, especially regarding the node where a page should be mapped to. On modern systems with large caches, cache misses might indicate that the page is not used frequently from a node, which can imply that a page should *not* be migrated to a node with lots of cache misses to that page. Furthermore, this proposal has similar drawbacks as LaRowe's, such as the lack of balance data mapping and thread mapping policy, as well as the overhead of maintaining coherence of replicated pages on modern systems.

More recent proposals in operating systems also use page faults for data mapping. Modern versions of the Linux kernel (since version 3.8) include the NUMA Balancing technique (CORBET, 2012b) for the x86_64 architecture. NUMA Balancing support was extended to the PowerPC architecture in kernel version 3.14. A previous proposal with similar goals was AutoNUMA (CORBET, 2012a). NUMA Balancing uses the page faults of parallel applications to detect memory accesses and performs a sampled next-touch strategy. Whenever a page fault happens and the page is not located on the NUMA node that caused the fault, the page is migrated to that node. However, this mechanism keeps no history of accesses, which can lead to a high number of migrations, and it performs no thread mapping to improve the gains of the data mapping.

Current research uses a history of memory accesses to limit unnecessary migrations and perform thread mapping. The Carrefour mechanism (DASHTI et al., 2013) has similar goals as NUMA Balancing, but uses Instruction-Based Sampling (IBS) (DRONGOWSKI, 2007), available in recent AMD architectures, to detect the memory access behavior and keeps a history of memory accesses to limit unnecessary migrations. Additionally, it allows replication of pages that are mostly read. However, the authors need to use the sampled accesses to predict if a page will be written to, as these writes have a large overhead due to the coherence, and the OS keeps only very coarse-grained information about the write permissions to pages (BASU et al., 2013). To limit the runtime overhead, Carrefour limits

its characterization and data mapping to 30,000 pages (corresponding to about 120 MByte of main memory with 4 KByte pages), which limits its applicability to small applications. The authors suggest that Carrefour could be ported to Intel-based architectures via the Precise Event Based Sampling (PEBS) framework (LEVINTHAL, 2009).

Blagodurov et al. (BLAGODUROV et al., 2010) discuss contention on NUMA architectures and identify thread migrations between NUMA nodes as detrimental to performance. Awasthi et al. (AWASTHI et al., 2010) propose balancing the memory controller load by dynamically migrating pages from overloaded controllers to less loaded ones. They estimate load from row buffer hit rates gathered through simulation. These techniques depend on particular hardware features and are therefore difficult to apply on new systems. In Section 7.5, we compare Carrefour and NUMA Balancing to our proposal, since they are the two most recently-proposed mechanisms and can be applied to current hardware architectures.

7.2.2 Compiler-Based and Runtime-Based Mapping

Other mechanisms perform mapping in user space, most of them on the compiler or runtime level. All these techniques have in common that they only have knowledge about a single application. Since systems usually execute several applications at the same time, their mapping decisions can interfere between them or with the OS, which limits their applicability to cases where execution of only a single application can be guaranteed. Some techniques, such as the previously mentioned ForestGOMP (BROQUEDIS et al., 2010a), are limited to specific parallelization libraries, such as OpenMP. ForestGOMP performs data mapping by grouping OpenMP threads according to their affinity and then allocate the data they access on the same NUMA node where they will execute. The authors also mention that the data mapping can be improved with the help of hardware counters, but do not describe this idea in detail.

Majo et al. (MAJO; GROSS, 2012) identify memory accesses to remote NUMA nodes as a challenge for optimal performance and introduce a set of OpenMP directives to perform distribution of data. The best distribution policy has to be chosen manually and may differ between different hardware architectures. Ogasawara et al. (OGASAWARA, 2009) propose a data mapping method for object-oriented languages that use garbage collection. During garbage collection, their mechanism collects information about which thread accesses an object frequently, and uses this information to migrate objects to nodes to reduce the number of remote NUMA accesses. This technique is limited to languages and runtime environments that use garbage collection, such as Java. Piccoli et al. (PICCOLI et al., 2014) use compiler-inserted code to predict memory access behavior in parallelized loops and use the prediction to migrate pages before the loop is executed. No thread mapping or balancing operations are performed. Nikolopoulos et al. (NIKOLOPOULOS et al., 2000b; NIKOLOPOULOS et al., 2000a) propose an OpenMP library that gathers information about thread migrations and memory statistics of parallel applications to migrate pages between NUMA nodes when threads are migrated.

7.2.3 Hardware-Based Mapping

Another category of mechanisms use statistics generated from hardware counters to guide mapping decisions. These techniques are generally limited to specific hardware architectures. Marathe et al. (MARATHE; MUELLER, 2006) present an automatic page placement scheme for NUMA platforms by tracking memory addresses from the performance monitoring unit (PMU) of the Intel Itanium processor. The profiling mechanism is enabled only during the start of each application due to the high overhead and therefore loses opportunities to handle changes during the rest of the execution.

Tikir et al. (TIKIR; HOLLINGSWORTH, 2008) use UltraSPARC III hardware counters to provide information for the data mapping, but do not perform thread mapping. Their proposal is limited to architectures with software-managed Translation Lookaside Buffers (TLBs), which is only a minority of current systems. The Locality-Aware Page Table (LAPT) (CRUZ et al., 2014b) is an extended page table that stores the memory access behavior for each page in the page table entry. This information is updated by the Memory Management Unit (MMU) on every TLB access. The operating system evaluates the behavior periodically to improve the thread and data mapping. This mechanism requires changes to the hardware, since current MMUs do not contain support for updating the page table in this way.

7.2.4 Manual Mapping

Most proposals for manual data mapping focus on giving tools to developers to make their application NUMA-aware or to suggest a particular mapping based on memory access traces. Libraries that support NUMA-aware memory allocation include libnuma (KLEEN, 2004; DREPPER, 2007) and MAi (RIBEIRO et al., 2009). With these libraries, data structures can be allocated according to the specification of the developer, such as on a particular NUMA node, or with an interleave policy. These techniques can achieve large improvements, but place the burden of the mapping on the developer and might require rewriting the code for each different architecture. An evolution of MAi, the Minas framework (RIBEIRO et al., 2010), optionally uses a source code preprocessor to determine data mapping policies for arrays. Antony et al. (2006) discuss data mapping improvements from explicit affinity management on UltraSPARC and AMD architectures, based on two NUMA benchmarks.

Dupros et al. (2010) perform an in-depth analysis of Ondes3D, and suggest allocating memory pages on the NUMA node that performs the most accesses to it with the help of the MAi library. We compare the performance improvements of this manual optimization to kMAF in Section 7.5.4. Previous research also uses memory access traces to perform data mapping. In Marathe et al. (2010) present an automatic page placement scheme for NUMA platforms by based on a hardware-assisted memory tracer that uses the performance monitoring unit (PMU) of Itanium-2. Similarly, Bolosky et al. (1992) single-step each instruction executed by the CPU to generate a trace.

Other proposals run entirely on the software level, through simulation (CRUZ et

al., 2011) or Dynamic Binary Instrumentation (DIENER et al., 2015; DIENER; CRUZ; NAVAUX, 2015). These techniques can be useful to determine the maximum gains that can be achieved with mapping policies, but are not applicable in general due to their substantial overhead and the fact that the access behavior might change with different input data and different numbers of threads, among others. We used such tracing techniques to implement an Oracle-based mapping that was presented in Chapter 5, and will compare this Oracle to kMAF in this chapter.

7.2.5 Summary of Related Work

An overview of the main related work in data mapping is shown in Table 7.1. Summarizing our analysis of related work, we conclude that kMAF is the first mechanism to have all the following important properties: (1) it handles thread and data mapping jointly, (2) it is based on the kernel level, which makes it independent of the parallelization API and supports multiple applications executing at the same time, (3) it requires no changes to runtime libraries, applications or the hardware, (4) it supports policies that focus on locality or balance, and (5) it requires no prior information about the memory access behavior.

Table 7.1: Overview of the main related work in data mapping. Each check mark indicates support for a desirable property.

Mechanism	Type	Metric	Thread mapping	Multiple applications	Balance	Locality	No hardware change	Hardware independent	No application change	No previous execution
Carrefour	Auto.	IBS		✓	✓	✓	✓		✓	✓
ForestGOMP	Auto.	Shared data	✓			✓	✓	✓	✓	✓
Piccoli et al.	Auto.	Shared data				✓	✓	✓	✓	✓
LAPT	Auto.	TLB accesses	✓			✓			✓	✓
Libnuma, MAi	Manual	Developer	✓			✓	✓	✓		✓
Marathe et al.	Auto.	PMU stats				✓	✓		✓	✓
NUMA Balancing	Auto.	Page faults		✓	✓		✓	✓	✓	✓
LaRowe et al.	Auto.	Page faults		✓		✓	✓		✓	✓
Tikir et al.	Auto.	PMU stats		✓	✓	✓	✓		✓	✓
kMAF	Auto.	Page faults	✓	✓	✓	✓	✓	✓	✓	✓

7.3 kMAF: The Kernel Memory Affinity Framework

The *kernel Memory Affinity Framework (kMAF)* uses the paging-based virtual memory implementation of modern computer architectures to analyze the memory access behavior of parallel applications during execution and uses the generated information to perform an improved thread and data mapping. This section presents kMAF, discusses its implementation and examines its overhead.

7.3.1 Overview of KMAF

kMAF consists of four parts, which are shown in Figure 7.1.
1. Detect memory access behavior. The memory access behavior of the parallel application is detected by tracing its page faults. Extra low-latency page faults are inserted throughout the execution to improve the detection accuracy.
2. Storage and analysis of the behavior. The memory addresses and thread IDs of the page faults are stored in two tables on the page and sub-page granularity, for data and thread mapping, respectively. For thread mapping, kMAF also maintains a sharing matrix.
3. Perform thread mapping. The sharing matrix is evaluated periodically with a mapping algorithm to determine if threads should be migrated between cores.
4. Perform data mapping. The access behavior to a page is analyzed during each page fault to determine if a page should be migrated between NUMA nodes.

7.3.2 Detecting Memory Access Behavior

As for CDSM, detecting the memory access behavior of parallel applications efficiently is a critical step of kMAF, since it determines the accuracy and overhead of the information and has a direct impact on the performance improvements that can be achieved. As memory accesses are usually performed directly by the hardware, without a notification to the OS, gathering information about them in a portable way is challenging. In most architectures, the OS is notified about a memory access only when a page fault happens. We make use of this fact in kMAF by observing the page faults of the parallel application. We use a similar mechanism as presented in Chapter 6 to determine the memory access behavior.

7.3.2.1 Observing Memory Accesses via Page Faults

Whenever information about a virtual-to-physical address mapping is missing from the page table for a particular application, or when that information is invalid, the CPU signals a page fault to the operating system, including information about the faulting virtual address. The thread ID that caused the fault is determined by the OS. By tracking this information, kMAF can detect the memory access behavior.

Since the faulting address is the full address, not just the address of the page, different granularities can be applied for the detection, such as the cache line size or the

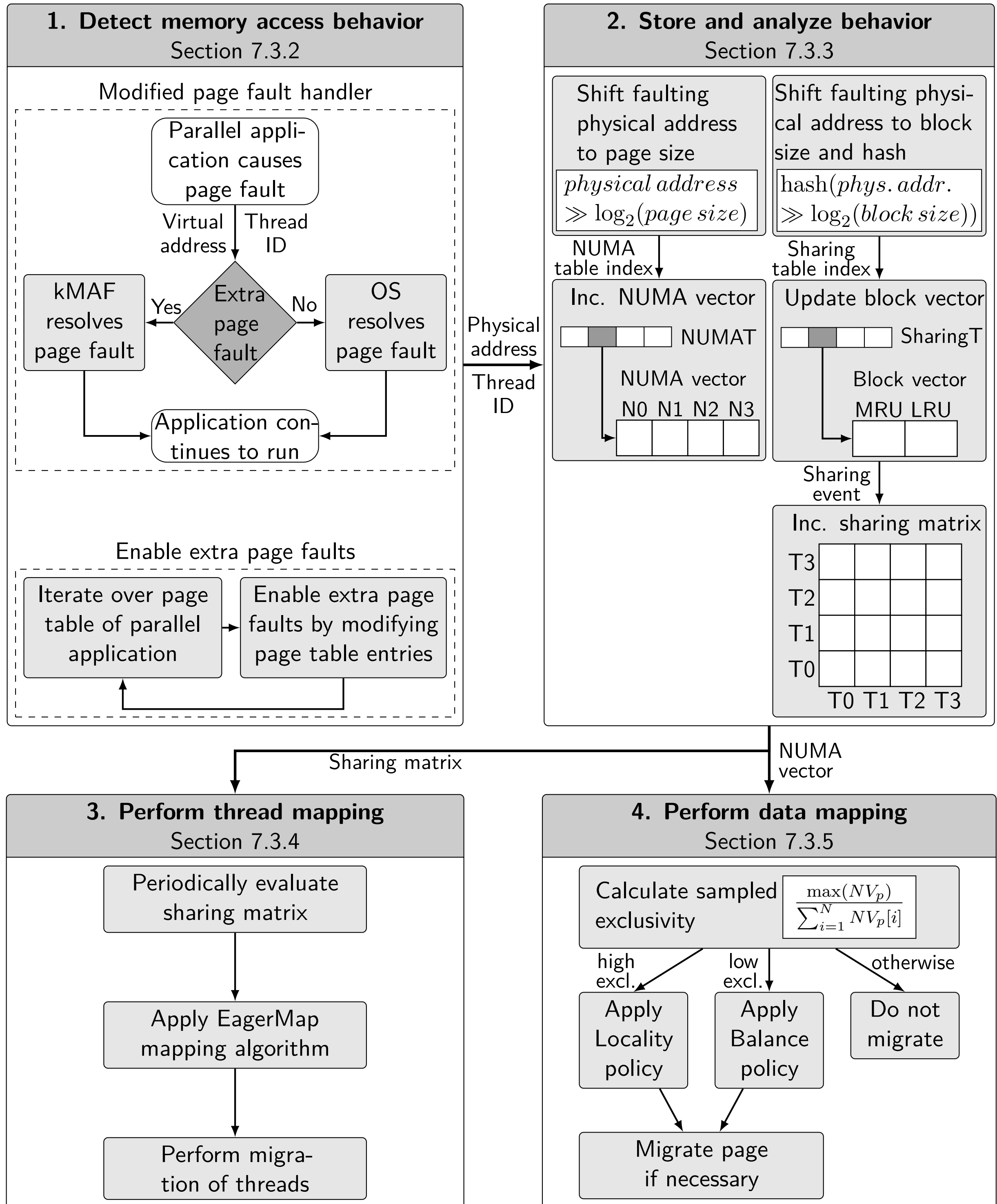


Figure 7.1: Overview of the kMAF mechanism. Data structures are shown for a NUMA machine with 4 nodes (N0–N3) and an application consisting of four threads (T0–T3).

specific page size of the architecture. For this reason, the detection is also not restricted to the page size that the hardware uses. This fact is important for the thread mapping due to its focus on improving the usage of caches. To support multiple running applications or processes that do not share a page table[1], kMAF converts the virtual address to the physical address for the detection, which is unique in the system.

7.3.2.2 Increasing Detection Accuracy with Extra Page Faults

An important aspect of the detection is that in normal circumstances, only one page fault happens for each page. After the page fault has been resolved by the OS by inserting a translation into the page table, subsequent accesses to the page will not generate further page faults[2]. To overcome this restriction and to increase the accuracy of the detection, kMAF inserts extra page faults during the execution of a parallel application, such that multiple faults can happen for the same page. Page faults are inserted in all memory areas that store data, but not for code segments.

To insert the additional faults, kMAF periodically iterates over the page table of the parallel application and modifies entries in such a way that the next memory access to the page will generate another page fault. The extra faults can be inserted in different ways. In most architectures, each page table entry contains a *present* bit which indicates if an entry is valid. To insert an extra page fault, kMAF clears this bit. As the extra page faults do not indicate missing information in the page table, no expensive operation (such as an allocation of a new page) has to be performed. The extra fault can be resolved by kMAF itself, by setting the present bit, without the use of other kernel routines. This reduces the overhead of these extra faults. To accurately characterize applications with different memory usages, kMAF scales the number of extra faults with the memory consumption of the application, as a percentage of the total of number pages that the application uses.

7.3.3 Storage and Analysis of the Detected Behavior

On each page fault, two pieces of information are gathered from the detection, the complete physical address that caused the fault and the thread ID. kMAF updates the memory access pattern with this data in two ways, on a per-page granularity for data mapping and on a finer granularity for thread mapping.

7.3.3.1 Information for Data Mapping

Since data mapping operates on the page level, information is stored about the memory access pattern of each page from the NUMA nodes. For each page that is allocated (that is, which is accessed at least once), kMAF maintains a *NUMA vector (NV)*, where

[1] Multithreaded applications running on Linux share a page table.

[2] Multiple page faults can happen for a single page in case multiple threads access the page for the first time in parallel.

each element $NV_p[n]$ stores the number of page faults to page p from node n. On each page fault, the physical address is shifted to the page size with Equation 7.1.

$$index = physical\,address \gg \log_2(page\,size) \tag{7.1}$$

The resulting *index* is used to access a *NUMA Table (NUMAT)* which stores the NUMA vectors for the pages. The NV_p of the page is then incremented for the NUMA node on which the thread that caused the page fault is executing.

7.3.3.2 Information for Thread Mapping

For thread mapping, a different type of information is necessary. Thread mapping has the goal of optimizing the usage of the caches in the system by mapping threads that share data on cores that share caches, in addition to increasing the memory access locality for pages that are shared between threads (DIENER et al., 2014). For this reason, the page granularity, which is many times larger than the cache line size, is not sufficient to accurately detect the sharing behavior. Furthermore, since the thread mapping algorithm requires only an estimation of the amount of sharing between all pairs of threads (represented as a *sharing matrix*), storing information about every page is not necessary.

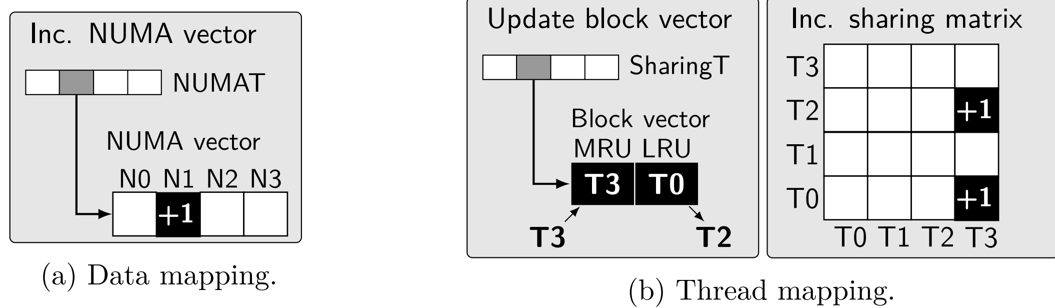
With these considerations, we implemented a simplified sharing detection for kMAF, similar to the one used by CDSM. We split the memory into *memory blocks* of a configurable size, with a default size of 256 bytes. The sharing detection is based on a hash table, which we call *sharing table*, that stores the IDs of the threads that recently accessed different memory blocks. The index of the hash table element is calculated with Equation 7.2. We use the default hash function provided by the Linux kernel.

$$index = \text{hash}(physical\,address \gg \log_2(block\,size)) \tag{7.2}$$

For each block, kMAF stores a *block vector (BV)*, which contains a short list of the threads that previously accessed the block. In the default configuration shown in Figure 7.1, the list has two elements, MRU and LRU. Whenever a thread accesses the block, its ID is inserted at the MRU position, shifting a previous ID to the LRU position. The element at the LRU position leaves the vector. kMAF increments the sharing matrix at the column of the thread ID and the rows of the previous threads that accessed the block.

7.3.3.3 Example of the Update of Data Structures

Figure 7.2 contains an example of the update of the data structures during a page fault. Consider an architecture with 4 NUMA nodes, in which thread 3, executing on node 1, causes a page fault in a block that has been previously accessed by threads 0 and 2, with thread 0 performing the most recent access. kMAF then increments the NUMA vector of the page that was accessed in position 1, representing node 1 (Figure 7.2a). The block vector of the block that was accessed contains two threads. Therefore, the oldest thread at the LRU position is removed from the BV (T2), T0 is shifted from the MRU to



(a) Data mapping.

(b) Thread mapping.

Figure 7.2: Example of the update of kMAF's data structures. Consider that thread 3 (executing on NUMA node 1) causes a page fault in a block that has been accessed by threads 0 and 2 before.

the LRU position, and T3 is inserted at the MRU position. Then, the sharing matrix is incremented for thread 3 with all the threads that were stored in the BV (Figure 7.2b).

7.3.4 Thread Mapping

To calculate the thread mapping, we use the same mechanism based on the EagerMap algorithm that was described in Section 6.3.3, with the same communication matrix and hardware description as input, as well as the same thread mapping interval of 200 ms.

7.3.5 Data Mapping

In contrast to the thread mapping, where the global communication was evaluated periodically, the memory access behavior to pages is evaluated locally (for each page) during every page fault. This is done for two reasons. As a parallel application can use millions of pages, performing the data mapping for all pages at the same time is not practical as it would lead to a substantial overhead for the calculation of the mapping and the page migrations. Furthermore, by analyzing the access behavior and performing eventual migrations during the page faults, there is no need for an additional context switch from the application to the kernel.

On the first memory access to a page, kMAF maintains the traditional first-touch semantics and allocates the page on the NUMA node that performs the first access to it. On subsequent accesses, the data mapping is performed in three steps during the page fault. First, the sampled exclusivity for the page is calculated from the NUMA vector, which describes if a page is (mostly) accessed from a single NUMA node, as discussed in Section 4.2. The exclusivity is used to apply a locality-based or balance-based mapping policy to the page. The page is then migrated to the node returned by the mapping policy. These steps are described in the next subsections.

7.3.5.1 Sampled Exclusivity and Policy Selection

To determine which page migration policy should be a applied to a certain page p, we calculate the *sampled exclusivity* $Excl_{sample}$ of the page, as shown in Equation 7.3, where NV_p is the NUMA vector of page p, N is the total number of NUMA nodes in the system, and the max function returns the maximum value of its argument. Pages with a high exclusivity are accessed mostly from a single NUMA node and are therefore more suitable for locality-based mapping.

$$Excl_{sample}[p] = \frac{\max(NV_p)}{\sum_{i=1}^{N} NV_p[i]} \tag{7.3}$$

The mapping policy is then selected with Equation 7.4, based on the sampled exclusivity.

$$MapPolicy[p] = \begin{cases} Locality, & \text{if } Excl_{sample}[p] > min_{Loc} \\ Balance, & \text{if } Excl_{sample}[p] < max_{Bal} \\ none, & \text{otherwise} \end{cases} \tag{7.4}$$

For pages with a high exclusivity ($> min_{Loc}$), which can benefit from increasing the locality of memory accesses, a Locality policy is applied. If the exclusivity is low ($< max_{Bal}$), kMAF applies a Balance policy, since this page can not benefit from a better locality. Otherwise, no mapping is performed and kMAF returns execution to the application. This process resembles our Oracle-based mapping mechanism that was discussed in Chapter 5.

7.3.5.2 Locality Policy

If the Locality policy is selected, kMAF applies a mapping filter to reduce the number of page migrations in case the memory access behavior to the page changes quickly. This filter is expressed in Equation 7.5, where NV_p is the NUMA vector of page p, and the max / $\max_2$ functions return the largest and second largest value of the vector, respectively.

$$Mig_{Local}[p] = \begin{cases} yes, & \text{if } \max(NV_p) \geq 2 \times \max_2(NV_p + 1) \\ no, & \text{otherwise} \end{cases} \tag{7.5}$$

The idea of the equation is to prevent early migration of pages, during application initialization when the access pattern is not established yet and might change very quickly. After the pattern stabilizes, the equation enables quick migration of pages.

The actual NUMA node where a page should be migrated to is calculated with Equation 7.6, where the $\arg\max$ function returns the element with the highest value.

$$node[p] = \arg\max(NV_p) \tag{7.6}$$

7.3.5.3 Balance Policy

kMAF limits the amount of migrations between the Balance and Locality policies by only applying the Balance policy after the access pattern has stabilized, as expressed by Equation 7.7, where N is the number of NUMA nodes.

$$Mig_{Balance}[p] = \begin{cases} yes, & \text{if } \sum NVp > N \\ no, & \text{otherwise} \end{cases} \tag{7.7}$$

If this condition is fulfilled, kMAF balances the page by using an *Interleave* policy, which has shown good balancing results (see Section 5.3). The NUMA node for a page is calculated with the help of the address of a page, as shown in Equation 7.8, where $\text{addr}(p)$ is the virtual address of page p.

$$node[p] = (\text{addr}(p) \gg \log_2(page\,size)) \bmod N \tag{7.8}$$

Such an Interleave policy has two main advantages compared to more complex techniques: (1) Since the NUMA node is calculated directly from the page address, there is no need to store or iterate over the global state of the application (such as the NUMA nodes or NUMA vector of all pages) to determine the node to migrate to, reducing the overhead to calculate the mapping. (2) For the same reason, a page with a low exclusivity is migrated only once as long as its exclusivity remains low, reducing the number of unnecessary page migrations.

7.3.5.4 Example of Policy Behavior

To illustrate the interaction between the various data mapping policies of kMAF, consider an example with a system consisting of N=3 NUMA nodes, $min_{Loc} = 70\%$, $max_{Bal} = 60\%$ and a single page p with the address 2. Table 7.2 shows the memory access behavior of a parallel application for this scenario. The application executes with 1 thread per node. In the beginning, consider that p has not been accessed before. In total, the page is accessed 14 times, labeled *a*–*n* in the table. The behavior of the application can be separated into several access patterns.

The page is accessed first from node N0, and is allocated on this node due to the first-touch semantics, which are maintained by kMAF (step *a*). In the second phase of the application (steps *b*–*e*), page p is accessed exclusively from node N2. In steps *c* and *d*, the balance policy of kMAF is applied, but the page is not migrated, since Equation 7.7 is not fulfilled. The page exclusivity is increasing, and in step *e* the locality Equation 7.5 is fulfilled, such that page p is migrated to node N1.

After the migration, consider that the application enters a new phase, and all memory accesses are now performed from node N0. This indicates that the exclusivity of the page is decreasing again. In step *f*, the exclusivity is between the two limits min_{Loc} and max_{Bal}, and no policy is applied. In step *g*, kMAF applies the balance policy and migrates the page, since Equation 7.7 is fulfilled. The node chosen for the migration (N2) is determined with the modulo operation ($2 \mod 3 = 2$). Although N2 did not access that page at all, migrating it to this node increases the memory access balance in this

Table 7.2: Behavior of kMAF's data mapping policy, for a single page p and a NUMA machine consisting of 3 NUMA nodes. Consider that min_{Loc} equals 70%, max_{Bal} equals 60%, N=3, and the address of page p is 2. Steps *a*–*n* are discussed in the text.

	NUMA vector				kMAF	
Steps	N0	N1	N2	Excl.	Policy	Action
a) First memory access from N0	1	0	0	100%	First-touch	Allocate p on N0
b) memory access from N1	1	1	0	50%	Balance	—
c) memory access from N1	1	2	0	67%	Balance	—
d) memory access from N1	1	3	0	75%	Locality	—
e) memory access from N1	1	4	0	80%	Locality	Migrate p to N1
f) memory access from N0	2	4	0	67%	—	—
g) memory access from N0	3	4	0	57%	Balance	Migrate p to N2
h) memory access from N0	4	4	0	50%	Balance	—
i) memory access from N0	5	4	0	56%	Balance	—
j) memory access from N0	6	4	0	60%	Balance	—
k) memory access from N0	7	4	0	64%	—	—
l) memory access from N0	8	4	0	67%	—	—
m) memory access from N0	9	4	0	69%	—	—
n) memory access from N0	10	4	0	71%	Locality	Migrate p to N0

example. In steps *h*–*j*, the balance policy is also applied, but the page is not migrated since it is already located on node N2.

Between steps *k* and *n*, the exclusivity keeps on increasing, until min_{Loc} is passed in step *n*. In that step, p is migrated back to node N0. This example illustrates how kMAF reduces a page ping-pong between nodes. With more page accesses, multiple migrations of the same page are becoming more difficult, limiting the number of migrations of pages with very frequent changes in the access behavior.

7.3.5.5 Performing the Page Migration

To perform the page migration, kMAF needs the virtual address of the page, as well as the NUMA node where it should be migrated to. Before migrating, kMAF checks if the page is not already located on the node to be migrated to, and aborts the migration in that case. Otherwise, kMAF uses the `unmap_and_move()` function of the Linux kernel to perform the actual page migration to the requested NUMA node. The virtual address is not stored by kMAF, since it is available during the page fault and page migrations are only performed while the fault is handled.

7.3.6 Supporting Multiple Running Applications

One important advantage of kernel-based mapping solutions such as kMAF is their support for multiple applications that execute concurrently, in contrast to many user-space techniques. Since kMAF bases its detection of the memory access behavior on the physical addresses of the memory accesses, different applications executing at the same time do

not interfere with each other, maintaining the same detection accuracy as if only one application was running. For the thread mapping, kMAF maintains a single sharing matrix for the applications running, and applies the thread mapping algorithm with this matrix. The data mapping is performed without changes for multiple applications. In this way, multiple parallel applications are handled as one larger application and neither kMAF nor its configuration needs to be changed to support this case.

7.3.7 Implementation of KMAF

We implemented kMAF in the Linux kernel. The default page fault handler of the kernel was modified to enable the page fault tracking mechanism and to implement the data mapping. kMAF creates a kernel thread that enables extra page faults during execution. Another kernel thread implements the thread mapping mechanism. Table 7.3 contains an overview of the default configuration parameters of kMAF used in our experiments.

7.3.8 Overhead of KMAF

Since kMAF operates during the execution of parallel applications, it imposes a storage and execution time overhead.

7.3.8.1 Storage Overhead

kMAF needs to allocate memory for the NUMA table, sharing table and sharing matrix. The size of the NUMA table is calculated with Equation 7.9, where N is the number of NUMA nodes in the system and P is the number of pages that the application accesses. We store the NUMA vector with an element size of 1 byte, which can count up to 256 page faults per node and per page. Considering a system that consists of 4 NUMA nodes and a page size of 4 KByte, the storage overhead per page is $4/4,096 \approx 0.1\%$.

$$size(NUMAT) = N \times P \times 1\ byte \tag{7.9}$$

For the hash table that stores the sharing behavior, we store two thread IDs per block. Each ID has a size of 2 bytes to support up to 65,536 threads. In case of a hash conflict, the old block is overwritten. Equation 7.10 shows the overhead for the hash

Table 7.3: Default configuration of kMAF used in the experiments.

kMAF part	Configuration
Thread mapping	Hash table: 1 million blocks, each 256 byte large
	Mapping interval: 200 ms
Data mapping	min_{Loc}: 80%; max_{Bal}: $1/N \times 1.5 \times 100\%$
	NUMA vector: 1 Byte per page/node
Extra page faults	max. 10% of total pages/second

table, where *nBlocks* is the number of blocks that can be stored in it. In the default configuration with 1 million blocks, it has a size of 4 MByte.

$$size(SharingT) = 2 \times 2\ byte \times nBlocks \tag{7.10}$$

The sharing matrix used to calculate the thread mapping has a cell size of 4 byte. The total size of the matrix can then be calculated with Equation 7.11, where T is the number of threads of the parallel application. For an application with 1,024 threads, the matrix has a size of 4 MByte.

$$size(SharingMatrix) = T^2 \times 4\ byte \tag{7.11}$$

7.3.8.2 Execution Time Overhead

The runtime overhead consists of the time required to introduce extra page faults, resolve these faults, calculate the thread and data mappings, and the migrations. The complexity to introduce extra page faults increases linearly with the memory usage of the application. Resolving an extra page fault from kMAF has a constant time complexity. Our thread mapping algorithm has a complexity of $\mathcal{O}(T^3)$, where T is the number of threads of the parallel application. For data mapping, the time complexity is $\mathcal{O}(N)$, where N is the number of NUMA nodes. In Section 7.5, we will evaluate the runtime overhead on a running application.

7.4 Methodology of the Experiments

We experimented with the NAS-OMP and PARSEC benchmarks, with the *C* and *native* input sizes, respectively. We also discuss two scientific applications, Ondes3D and HashSieve. Experiments were performed on our three evaluations systems, *Itanium*, *Xeon*, and *Opteron*. The following mapping mechanisms were compared:

OS: The Linux OS forms the baseline for our experiments. We run an unmodified Linux kernel, and use its default first-touch mapping policy. The NUMA Balancing mechanism (CORBET, 2012b) is disabled in this configuration.

Compact: The compact thread mapping is a simple mechanism to improve memory affinity by placing threads with neighboring IDs (such as threads 0 and 1) close to each other in the memory hierarchy, such as on same cores, similar to options available in some OpenMP environments (INTEL, 2012b).

Oracle: To calculate an oracle-based thread and data mapping, we use a memory tracer based on the Pin Dynamic Binary Instrumentation (DBI) tool (LUK et al., 2005), as presented in Chapter 5.

NUMA Balancing: We use the NUMA Balancing mechanism (CORBET, 2012b) of version 3.8 of the Linux kernel, as described in Section 7.2.1. The mechanism uses a sampling-based next-touch migration policy. Whenever a parallel application causes a page fault, the page is migrated to the NUMA node on which the thread is executing. NUMA Balancing is only supported on the *Xeon* and *Opteron* machines.

Carrefour: The Carrefour mechanism (DASHTI et al., 2013) was evaluated on the *Opteron* machine, since it requires hardware features that are available only on AMD architectures. Carrefour was executed with its default configuration.

kMAF: Our proposed mechanism, kMAF, was implemented in the Linux kernel and executed with the configuration presented in Section 7.3.7. The kernel is the kernel used for the OS configuration, with NUMA Balancing disabled.

7.5 Results

This section presents the experimental results of kMAF. We begin with the results of a single parallel application and multiple applications that are executing concurrently. We then evaluate the sensitivity of kMAF to the number of extra page faults and discuss its runtime overhead.

7.5.1 Single Applications

The results for the three machines when executing a single parallel application at a time are shown in Figures 7.3–7.6.

7.5.1.1 Itanium

The results for the *Itanium* machine are shown in Figure 7.3. Most of the NAS-OMP benchmarks benefit from an improved mapping, indicated by the substantial improvements compared to the OS. The highest improvements were achieved for the CG-OMP benchmark, of up to 65% with the Oracle policy. Due to the relatively low number of NUMA nodes (2) and the simple memory access pattern of most NAS-OMP benchmarks, performing only the Compact thread mapping already results in high speedups. For the majority of the NAS-OMP benchmarks, the results of kMAF are between the Compact and Oracle policies. Only two of the PARSEC benchmarks, Facesim and X264, are suitable for mapping on this architecture, with similar performance improvements for kMAF and the Oracle. For Ferret and X264, the Compact policy results in substantial performance losses compared to the OS. The geometric mean of the improvements of all benchmarks is 1.7%, 14.3%, and 9.4% for Compact, Oracle, and kMAF, respectively.

7.5.1.2 Xeon

Figure 7.4 shows the performance results for the *Xeon* machine. From the NAS-OMP benchmarks, the highest improvements were achieved for the SP-OMP benchmark, with similar improvements for the Oracle, NUMA Balancing and kMAF. Despite the good improvements for some benchmarks (DC-OMP, IS-OMP and SP-OMP), NUMA Balancing causes significant slowdowns for others that can not benefit from mapping (CG-OMP, LU-OMP, MG-OMP and UA-OMP). The lack of an access history causes unnecessary page migrations and increases the runtime overhead. The Compact thread mapping only has minimal improvements, indicating the importance of data mapping. Several PARSEC

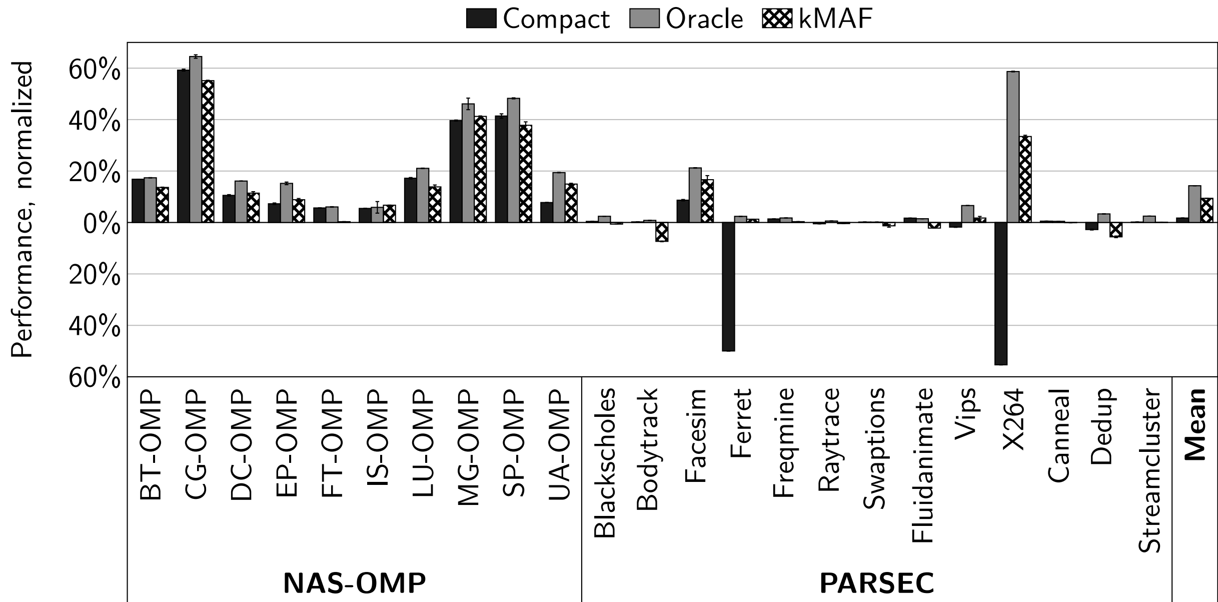


Figure 7.3: Performance improvements on *Itanium*, normalized to the results of the OS.

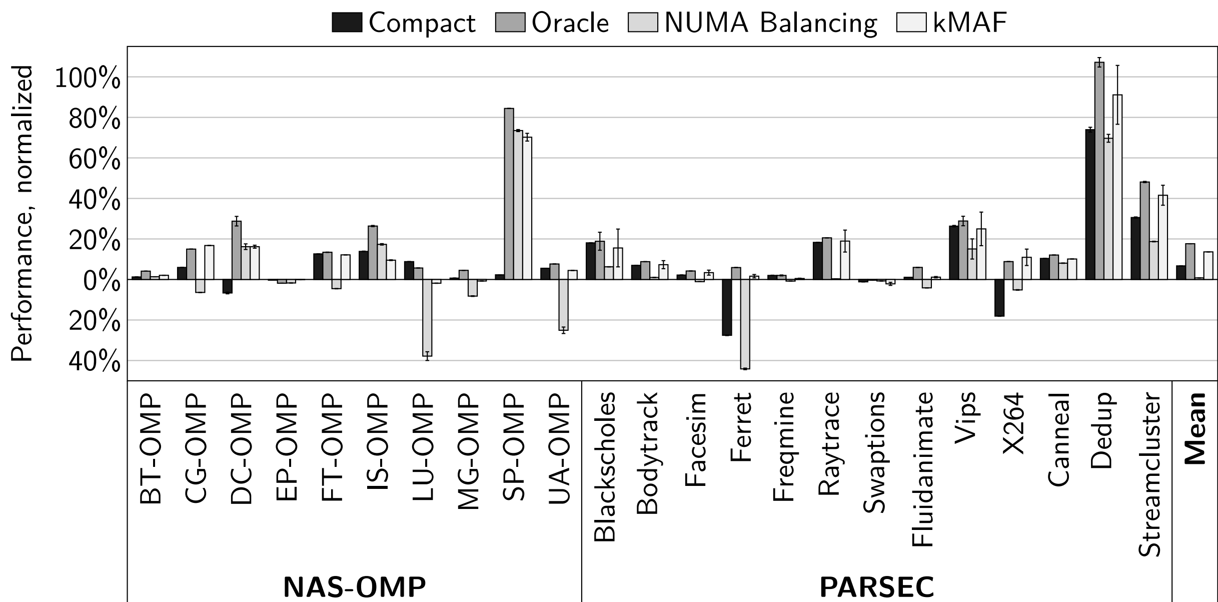


Figure 7.4: Performance improvements on *Xeon*, normalized to the results of the OS.

benchmarks benefit from mapping on *Xeon*. Dedup's performance was improved by up to 100%. Due to their lower memory usage, many PARSEC applications can benefit from thread mapping only, as evidenced by the results of the Compact mapping. For most PARSEC benchmarks, kMAF has the closest results to the Oracle of all policies. The geometric mean of the improvements of all benchmarks is 6.7%, 17.7%, 0.8%, and 13.6% for Compact, Oracle, NUMA Balancing, and kMAF, respectively.

The QuickPath Interconnect (QPI) traffic between processors, measured with the help of the Intel performance counter monitor (PCM), is shown in Figure 7.5. The reduction of the QPI traffic, with an average of 19.0% for kMAF, happened due to the

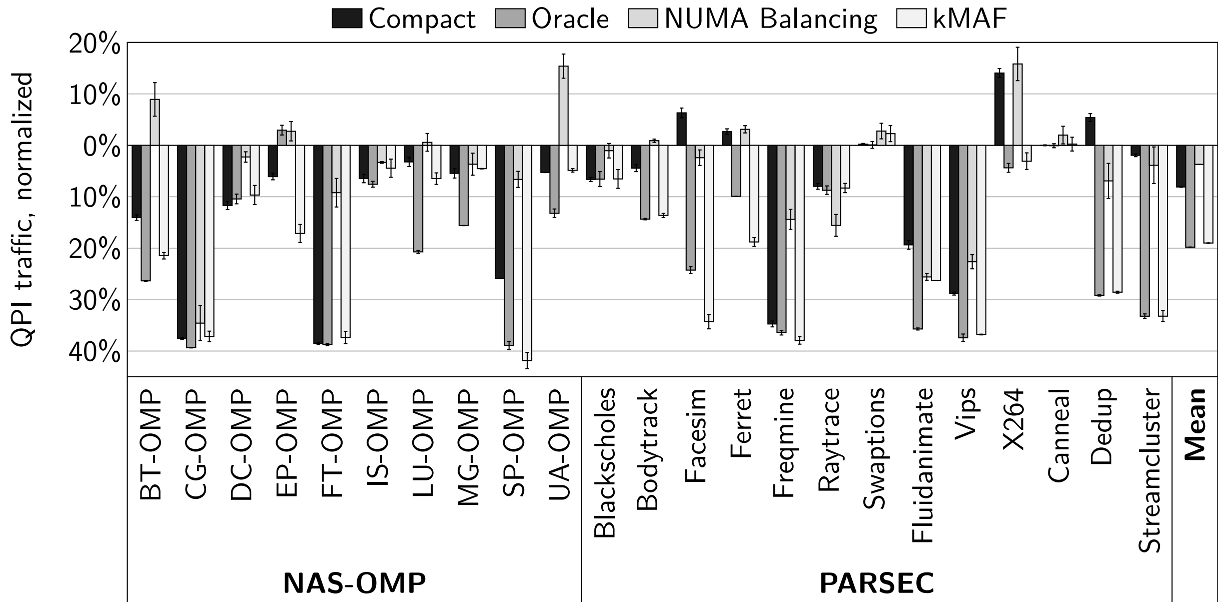


Figure 7.5: QPI traffic reduction on *Xeon*, normalized to the results of the OS.

optimized thread and data affinity, as the data is located more often in the local memory banks, removing the need to transfer data between processors. QPI reduction results correlate well with the performance improvements show in Figure 7.4.

7.5.1.3 Opteron

The results for the *Opteron* machine are shown in Figure 7.6. Overall, the highest performance improvements of the three machines were achieved on *Opteron*, due to its

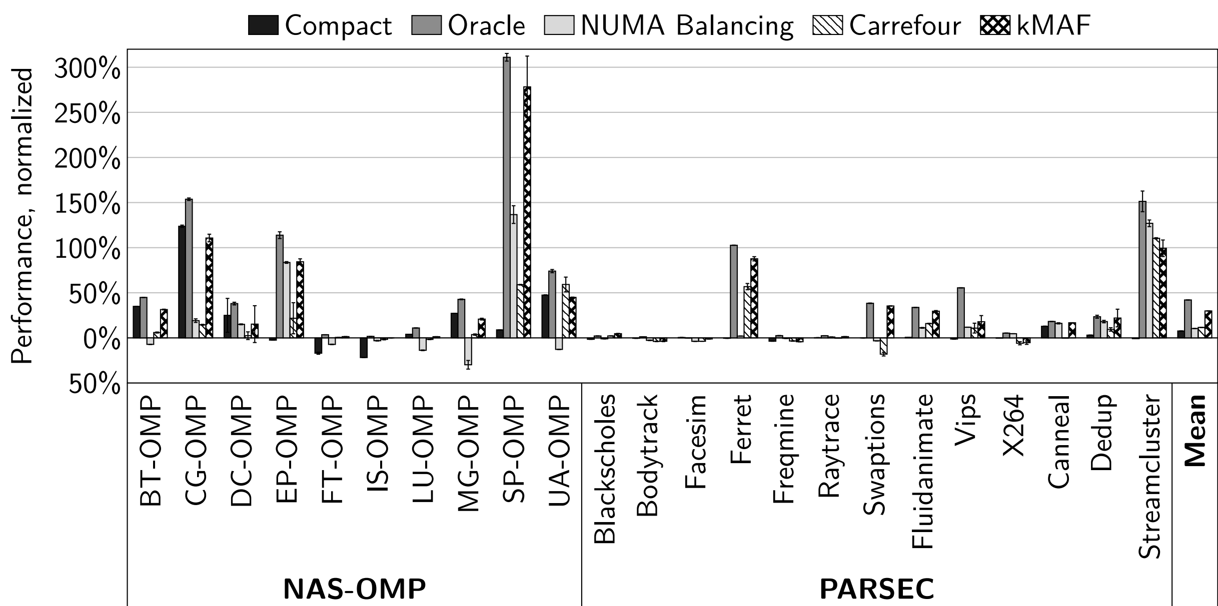


Figure 7.6: Performance improvements on *Opteron*, normalized to the results of the OS.

many NUMA nodes and high NUMA factor. Similar to *Xeon*, SP-OMP achieved the highest gains of the NAS-OMP benchmarks, of nearly 300% with kMAF. As before, kMAF had the closest results to the Oracle. Due to the high memory usage of the applications, the Carrefour mechanism had only low improvements since it limits itself to 30,000 pages (corresponding to 120 MByte with 4 KByte pages). The Compact thread mapping had negligible improvements except for CG-OMP. For the PARSEC benchmarks, Carrefour shows improvements that are closer to kMAF and the Oracle, since many PARSEC applications have a lower memory consumption (about 110 MByte in the case of Streamcluster, for example). However, both NUMA Balancing and kMAF have higher gains for most PARSEC benchmarks. The geometric mean of the improvements of all benchmarks is 7.7%, 42.0%, 11.6%, 10.4%, and 29.8%, for Compact, Oracle, Carrefour, NUMA Balancing, and kMAF, respectively.

7.5.1.4 Summary

Results show that simple ways to improve memory affinity, such as the Compact thread mapping, do not result in significant gains compared to the OS in most cases. Furthermore, mechanisms that keep no access history (NUMA Balancing) or limit the number of pages that they characterize (Carrefour) also do not result in optimal performance. Our kMAF mechanism provided the highest improvements overall, with substantial gains compared to the OS, close to the Oracle mechanism, on all three machines.

7.5.2 Energy Consumption

As discussed in Section 5.1.2.2, improved mappings can also result in higher energy efficiency. We evaluate the energy consumption improvements of the *Xeon* machine by using the Baseboard Management Controller (BMC), which exposes the energy consumption of the whole system through IPMI. The experimental methodology is the same as before.

The energy consumption, normalized to the results of the OS, is shown in Figure 7.7. As expected, the benefits are similar to the performance improvements, with applications that benefit more from mapping having higher energy savings. The highest improvements (of about 50%) were achieved for the SP-OMP and Dedup benchmarks. Similar to the performance results, the Compact thread mapping and NUMA Balancing did not achieve consistent improvements and actually reduce energy efficiency in several cases.

7.5.3 Multiple Applications

An important feature of kMAF is that it seamlessly supports multiple parallel applications that are executing concurrently, as discussed in Section 7.3.6, in contrast to solutions that operate in user space. We evaluate the support for multiple applications in this section.

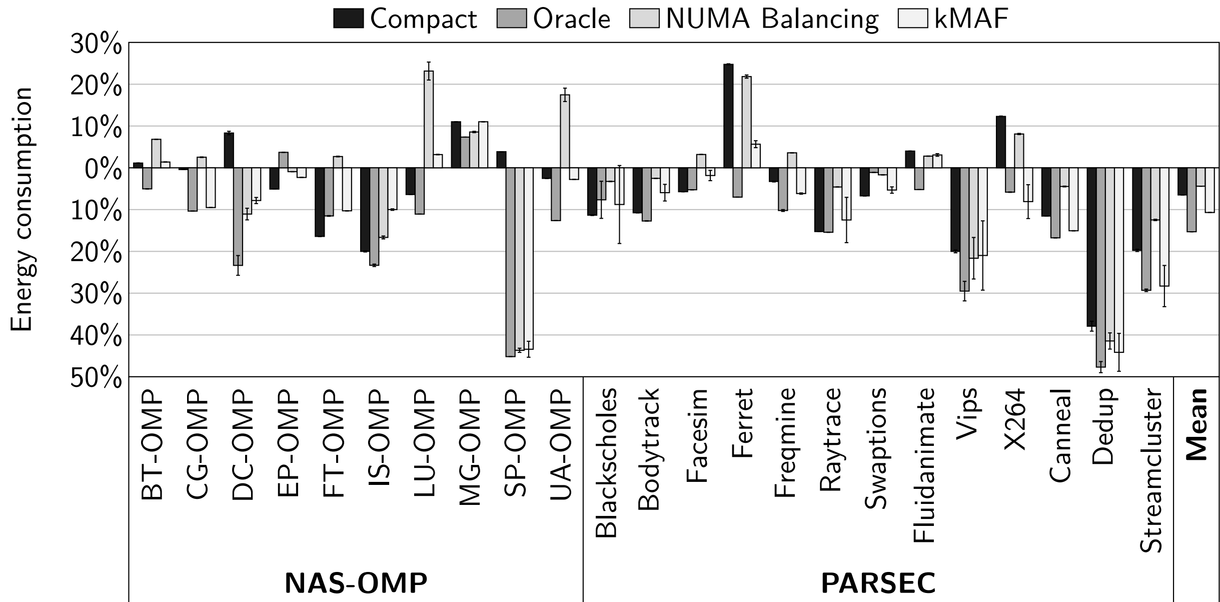
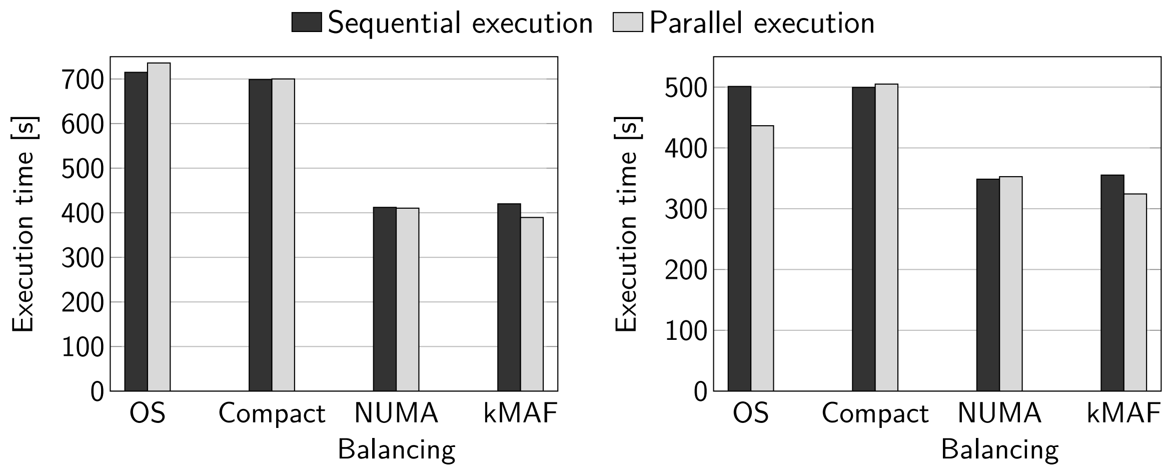


Figure 7.7: Energy consumption results on *Xeon*, normalized to the OS.



(a) SP-OMP+SP-OMP (C input). (b) SP-OMP (C input) and EP-OMP (D input).

Figure 7.8: Running multiple applications on the *Xeon* machine. All benchmarks execute with 64 threads. In the *sequential* configuration, applications run one after the other, and the total execution time is shown. In the *parallel* configuration, both applications start at the same time, and the execution time until both finish is shown.

7.5.3.1 Methodology

We selected two pairs of parallel applications, SP-OMP+SP-OMP, and SP-OMP+EP-OMP. SP-OMP is executed with the C input size, as before, but we use the D input size for EP-OMP to achieve comparable execution times and thereby maximize contention between the applications. All applications were executed with 64 threads each on the *Xeon* machine. Four mapping mechanisms (OS, Compact, NUMA Balancing and kMAF) and 2 different configurations (Sequential and Parallel) were compared. In the sequential configuration, the second application starts after the first one terminates. In this configuration, there

is less interference between applications and less contention for resources such as caches, interconnections, memory controllers and functional units. However, the utilization of resources may not be optimal in this case. For example, when a thread stalls while waiting for a memory request, another thread might make use of functional units at that time. In the parallel configuration, both applications are started at the same time, and we measure the time until both terminate. This configuration has the opposite characteristics of the sequential case, with higher contention for resources, but potentially more efficient usage.

7.5.3.2 Results

Figure 7.8 shows the execution time of the experiment. In the SP-OMP+SP-OMP case (Figure 7.8a), two memory-hungry applications are running, which leads to performance reductions for the OS in the parallel configuration. Since kMAF additionally performs thread mapping, it is able to reduce the overall contention and gains performance from the parallel execution as well. In the SP-OMP+EP-OMP case (Figure 7.8b), since EP-OMP is a CPU-bound application, it only competes for the functional units, not for memory accesses. For this reason, there is less overall contention, and even the OS can benefit from parallel execution, in addition to kMAF. The results for NUMA Balancing show that performing only data mapping is not sufficient for optimal results since the thread mapping can help to reduce contention.

7.5.4 Case Study: Ondes3D

As discussed in Section 1.3.2, manual changes to the source code of applications can be used to improve the memory access behavior in NUMA architectures. In this section, we compare kMAF to such a manual optimization of source code using the main numerical kernel extracted from the *Ondes3D* application (DUPROS et al., 2008). Ondes3D simulates the propagation of seismic waves. Aochi et al. discuss this numerical stencil in detail (AOCHI et al., 2013). In shared memory architectures, a common way to extract parallelism of this stencil is to exploit the triple nested loops of the three-dimensional problem. This allows a straightforward use of OpenMP directives.

Classic manual optimizations to such stencils rely on improvements using a first-touch data allocation policy. By exploiting the regular memory access pattern of the application, we can make sure that the memory accessed by each thread is allocated close to the thread (DUPROS et al., 2010). We experiment with two versions of Ondes3D, which differ in the way the input matrix is initialized in a parallel loop that iterates over all its elements. In the unmodified version, each thread initializes a part of the matrix that is determined by the OpenMP runtime environment. In the optimized version, we force each thread to initialize the part of the matrix that he will work on for the rest of the application. In this way, the optimized version has a better behavior considering a first-touch policy.

Table 7.4: Page usage statistics of Ondes3D for 4 NUMA nodes.

Metric	Ondes3D version	
	unmodified	optimized
Application exclusivity	98.73%	99.74%
First-touch memory access locality	26.63%	82.98%
First-touch memory access balance	241.29	37.16

7.5.4.1 Memory Access Behavior

The communication matrix of Ondes3D is depicted in Figure 7.9. Both versions of Ondes3D have very similar patterns with a clear maximum for neighboring threads. However, both versions also have a very low communication ratio (<1%), such that we can expect only limited gains from an improved thread mapping policy. Both the communication and load of Ondes3D are balanced.

Table 7.4 shows the page usage statistics of both Ondes3D versions. Both versions show a very high application exclusivity of more than 98%. The optimized version increases the exclusivity slightly by 1% due to the better initialization. The main improvements happen for the memory access locality and balance with the first-touch policy. First-touch locality was improved drastically due to the more correct first-touch in the initialization loop. The unmodified version of Ondes3D allocates the majority of data on NUMA node 1 and has therefore an unbalanced memory access behavior. This imbalance is reduced significantly with the optimized version. Despite these improvements, we can notice that the behavior is not perfect yet, which leaves some room for additional gains from other mapping techniques such as kMAF. We did not notice any significant dynamicity in communication or page usage behavior.

7.5.4.2 Results

Table 7.5 shows the absolute execution times of Ondes3D with different mapping mechanisms when executing the unmodified and the optimized code. For all mechanisms, the optimized version of Ondes3D is faster than the unmodified one. As predicted by our analysis of the communication behavior, the Compact thread mapping only improves performance slightly compared to the OS. For the optimized version, the Interleave and

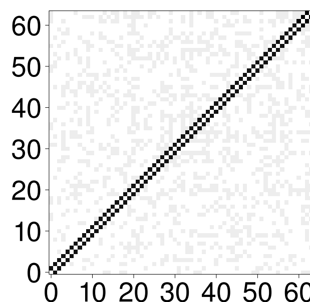


Figure 7.9: Communication pattern of both versions of Ondes3D.

Table 7.5: Execution time of the unmodified and optimized versions of Ondes3D on the *Xeon* machine with 64 threads.

	Ondes3D version	
Mapping technique	unmodified	optimized
OS	184.89 s	65.55 s
Compact	175.54 s	64.39 s
Interleave	79.17 s	73.03 s
Oracle	60.49 s	59.85 s
NUMA Balancing	93.20 s	70.38 s
kMAF	61.38 s	61.35 s

NUMA Balancing mechanism increase execution time compared to the OS, since the behavior is already well balanced. kMAF and the Oracle achieve the highest speedups, with improvements of more than 3× for the unmodified code. Differences between these two mechanisms are less than 2.5%. These results show that an automatic mechanism can achieve results that are very close to an Oracle, even for very large scientific applications. We can also see that even the optimized code can be improved by a better mapping mechanism, by 6.4% with kMAF.

7.5.5 Case Study: HashSieve

HashSieve is an example of a scientific application with a large, irregular memory access behavior (MARIANO; LAARHOVEN; BISCHOF, 2015), and therefore presents a contrast to Ondes3D, whose behavior was much more regular. HashSieve's main data structures are large hash tables that get accessed from all threads. Due to the hashing, the memory access behavior is extremely unstructured and changes on most memory accesses.

7.5.5.1 Memory Access Behavior

The communication matrix of HashSieve is depicted in Figure 7.10. We can see that the behavior is very homogeneous, as expected, and we predict few improvements from a thread mapping policy. Both the communication and load of HashSieve are balanced.

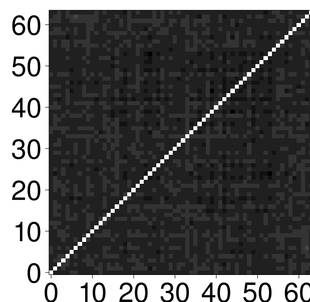


Figure 7.10: Communication pattern of HashSieve.

The page usage statistics of HashSieve shown in Table 7.6 also confirm our intuition

regarding the behavior. The application has a very low exclusivity of only 62.2% and its first-touch behavior has a reasonable locality of 85.0%. However, the memory access balance is low, with about 48% of memory accesses handled by NUMA node 1. These results indicate that HashSieve can benefit only slightly from a locality-based data mapping policy, but we expect higher gains from improving the memory access balance.

Table 7.6: Page usage statistics of HashSieve for 4 NUMA nodes.

Metric	Value
Application exclusivity	62.23%
First-touch memory access locality	85.04%
First-touch memory access balance	120.67

7.5.5.2 Results

Table 7.7: Execution time of HashSieve on the *Xeon* machine with 64 threads.

Mapping technique	Execution time
OS	354.97 s
Compact	331.77 s
Interleave	254.72 s
Oracle	243.19 s
NUMA Balancing	312.13 s
kMAF	267.49 s

The execution time of HashSieve with various mapping mechanisms is presented in Table 7.7. The Compact policy only results in small improvements, similar to Ondes3D. The Interleave and Oracle mappings result in the highest gains, reducing execution time by about 30% compared to the OS. Both NUMA Balancing and kMAF also significantly improve performance. Compared to the mappings with prior information, their gains are lower due to the later migration and higher runtime overhead. Nevertheless, kMAF gets the closest to the Oracle, within 7% of the execution time reduction.

7.5.6 Mechanism Sensitivity

kMAF's improvements are sensitive to the number of extra page faults. Increasing the number of extra page faults increases the detection accuracy, but also increases the overhead on the running application. Furthermore, kMAF's improvements also depend on the equation used to determine if a page should be migrated with a Locality policy (Equation 7.5). This section evaluates the impact of these two kMAF parameters.

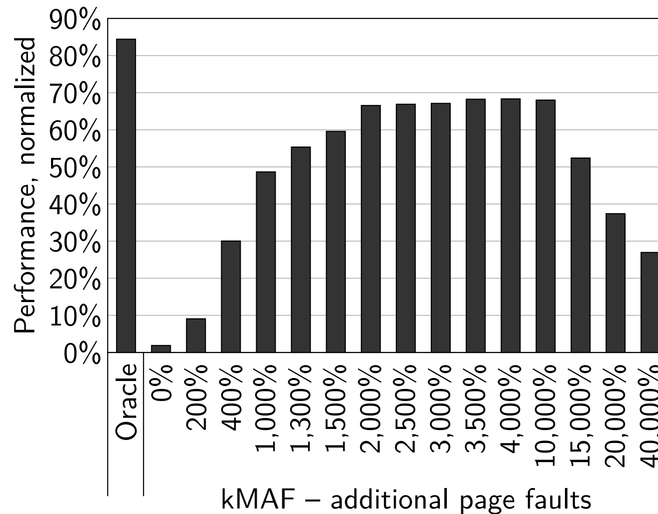


Figure 7.11: Performance improvements of kMAF when varying the number of additional page faults on the *Xeon* machine, with the SP-OMP benchmark (C input size). All values are normalized to the OS. The percentages on the x-axis indicate the number of additional page faults compared to the baseline without additional faults.

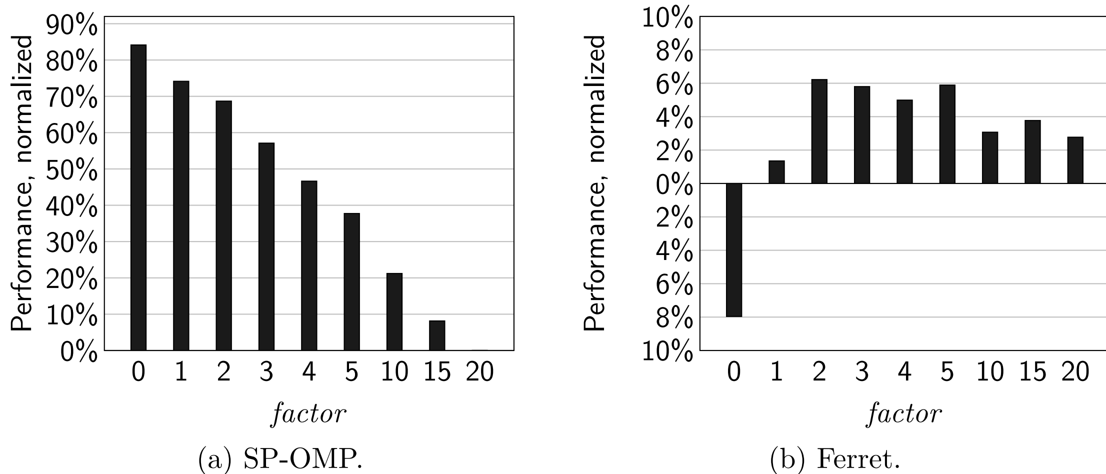
7.5.6.1 Sensitivity to the Number of Extra Page Faults

We evaluate the impact of the number of extra page faults on the performance gains of kMAF with the SP-OMP benchmark from NAS-OMP on the *Xeon* machine, comparing the results to the Oracle mapping and the OS.

Figure 7.11 shows the results of kMAF when varying the number of extra faults. Even with 200% extra faults, only small improvements can be achieved. The gains reach a maximum at 2,000% – 4,000% extra faults, becoming close to the Oracle mechanism. The default configuration of kMAF used in this thesis generates about 2,000% extra faults. Increasing the number of faults beyond 4,000% reduces the application performance due to the increasing overhead. However, even with 40,000% extra faults (that is, 400 faults per page on average), performance is still substantially higher than the OS.

7.5.6.2 Sensitivity to the Locality Formula

kMAF determines the NUMA node with a locality policy using Equation 7.5. This equation is reproduced below as Equation 7.12 in a slightly modified form. The purpose of the equation is to prevent excessive page migrations between nodes for pages with a dynamic access pattern. Here, the *factor* variable is used to limit these migrations, as discussed in Section 7.3.5.2. *factor* has a default value of 2, which is used in our experiments. Higher values of *factor* make it harder to move a page multiple times between nodes, but might lose more opportunities for improvements since pages are migrated later. In this section, we evaluate the influence of this factor on two applications, SP-OMP and Ferret. We use a Compact thread mapping to eliminate thread migrations and focus only on the data mapping. Several values of *factor* are evaluated: 0 (which might migrate pages on every page fault, similar to the NUMA Balancing mechanism, which keeps no access



(a) SP-OMP.

(b) Ferret.

Figure 7.12: Performance improvements when modifying the *factor* of the locality formula on the *Xeon* machine.

history), 1, 2 (default), 3, 4, 5, 10, 15, and 20. Apart from the changing the *factor*, we maintain kMAF at its default configuration.

$$Mig_{Local}[p] = \begin{cases} yes, & \text{if } \max(NV_p) \geq factor \times \max_2(NV_p + 1) \\ no, & \text{otherwise} \end{cases} \tag{7.12}$$

The performance compared to the OS for the two benchmarks on the *Xeon* machine are shown in Figure 7.12. For SP, a lower *factor* always results in higher improvements. This indicates that the memory access behavior of SP-OMP is very structured, and explains the slightly higher gains of NUMA Balancing for this application. For Ferret, the results show a different tendency. With low values for *factor*, performance is actually reduced compared to the OS. When increasing the *factor*, performance increases until reaching a maximum improvement between a value of 2–5. This behavior explains why the NUMA Balancing mechanism resulted in performance reductions for this benchmark.

7.5.7 Performing Thread and Data Mapping Separately

In Section 1.2.2, we discussed why thread mapping is a requirement for data mapping and evaluated the joint improvements from both types of mappings in a simulator in Section 5.1. To evaluate the influence of thread and data mapping on a real machine, we executed kMAF only with the thread mapping and data mapping part on the *Xeon* system. Three configurations were evaluated: thread and data mapping managed by the OS (baseline), kMAF thread mapping + OS data mapping, OS thread mapping + kMAF data mapping. All other parameters are the same as for our main experiments.

Figure 7.13 shows the execution time for the three configurations, normalized to the baseline. For most benchmarks, the improvements from the thread mapping only are higher than for the data mapping only. On average, data mapping reduced execution time by 2.7%, thread mapping by 5.9%. It is important to note that the improvements from performing

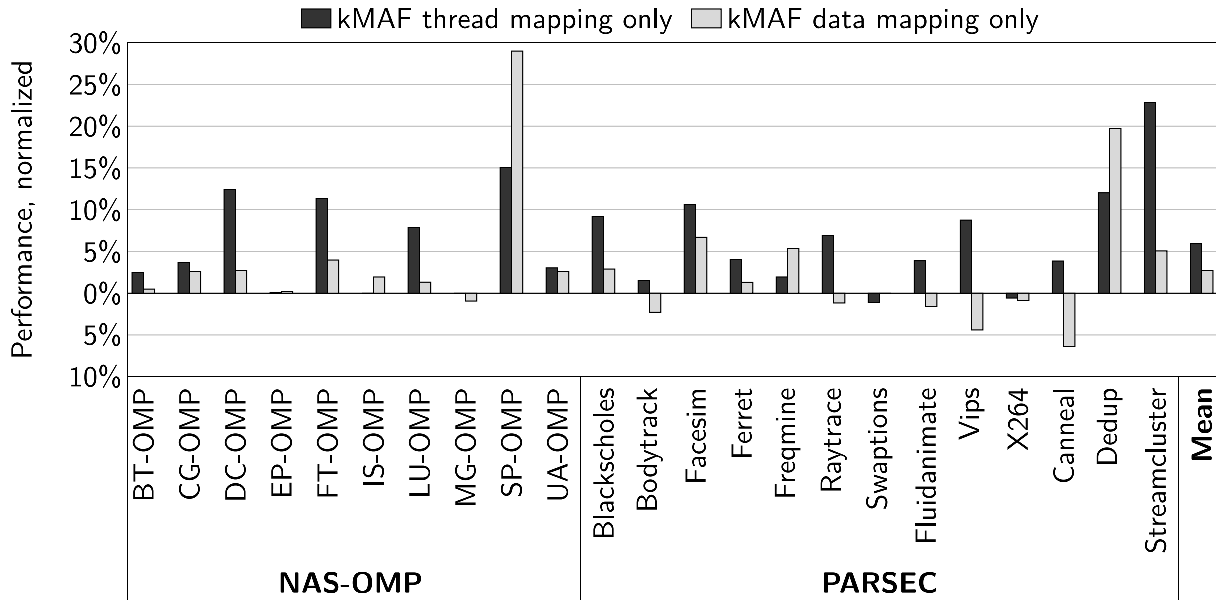


Figure 7.13: Performance improvements when running kMAF with only thread or data mapping on the *Xeon* machine, normalized to the OS.

thread and data mapping jointly, where kMAF achieved an average improvement of 13.6% as shown in Section 7.5.1.2, are higher than the sum of improvements when performing the mapping separately. This shows that integrated mechanisms are necessary for optimal results.

7.5.8 Performance Improvements with Larger Pages

As discussed in Sections 1.1.2 and 4.4.1.1, large pages present challenges for data mapping, since the granularity of migration decisions is increasing. However, large pages result in a more efficient execution due to less and more efficient page faults (ARCANGELI, 2010), and less TLB misses (BASU et al., 2013). We evaluate the impact of large pages on kMAF with the help of the Transparent Huge Page support of the Linux kernel (ARCANGELI, 2010), which allows applications to benefit from large pages automatically without changing the applications themselves. We selected the SP-OMP benchmark and execute it with the OS mapping and kMAF with two page sizes, small (4 KByte) and large pages (2 MByte). The same kMAF configuration was used in both cases.

The results of this experiment are shown in Figure 7.14. We measured execution time, data TLB store and load misses, as well as page faults, with the Linux perf tool (MELO, 2010). By only enabling large pages and letting the OS handle thread and data mapping, performance is already improved substantially, by 36%. However, improvements are still higher with kMAF, even with small pages (71%). Running kMAF with large pages only increases performance slightly, to 89% compared to the OS with small pages and 39% with the large pages. These results confirm our intuition that larger pages reduce the impact of data mapping, reducing the improvements by half. Nevertheless, data mapping still remains important.

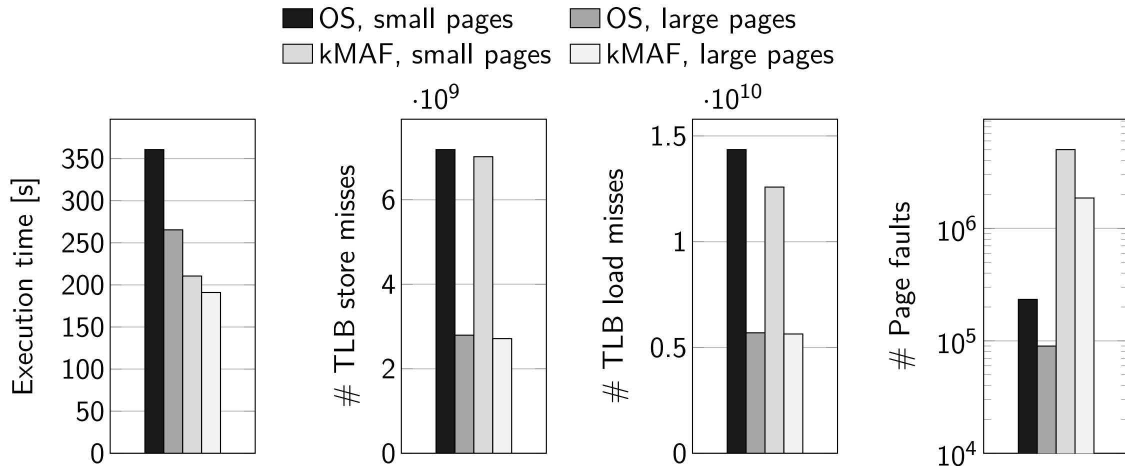


Figure 7.14: Performance results of SP-OMP when running with small (4 KByte) and large pages (2 MByte).

The larger pages result in much fewer TLB misses, about 60% less, both for store and load memory accesses. kMAF has a very similar TLB miss behavior as the OS, with very small improvements. Page faults with 2 MByte pages are reduced by about 50% with the OS. As before, kMAF increases the number of page faults compared to the baseline, but maintains a similar reduction of extra faults with larger pages.

7.5.9 Overhead of KMAF

Since it operates during the execution of parallel applications, kMAF imposes a runtime overhead. Figure 7.15 shows the number of execution cycles of a single event of each type of operation that kMAF performs, measured using the time stamp counter (TSC) on the *Xeon* machine. For each type of operation, the box plot shows the maximum, upper quartile, median, lower quartile, and minimum value. The median values are also shown above each category. We can confirm that the overhead of an extra page fault, which consists of the time to create and resolve it, is significantly lower than the overhead of the normal faults. In absolute terms, performing the page and thread migration and calculating the thread mapping have the highest computational demand. However, these are infrequent operations.

To evaluate the combined performance overhead during the execution of an application, we measure the total overhead of each category when executing the SP-OMP benchmark, which had the highest overhead in our experiments. Table 7.8 contains the number of events for each event type, absolute time spent for each event, and the overhead in % of the total execution time of SP-OMP. The results show that the overhead on the application is dominated by three categories, the extra page faults, the update of data structures and the page migrations. The total overhead corresponds to 2.4% of the total execution time of SP-OMP.

The total overhead for the evaluated benchmarks is presented in Table 7.9. Over

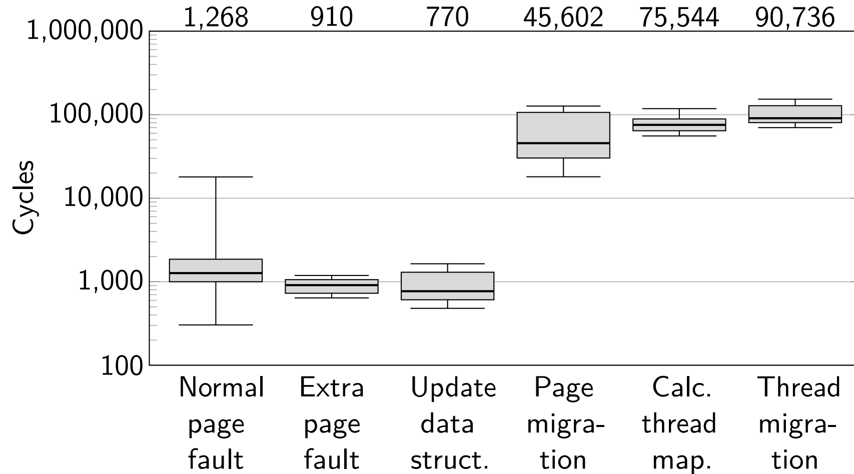


Figure 7.15: Number of execution cycles of each kMAF operation on the *Xeon* machine. Numbers on the top show the median value for each type.

Table 7.8: Overhead of the SP-OMP benchmark, measured on the *Xeon* machine.

Value	Normal page faults	Extra page faults	Update data struct.	Page migr.	Calculate thread mapping	Thread migr.
Number of events	235,520	4,828,171	5,063,691	176,194	3,487	370
Absolute time	0.15 s	2.19 s	1.95 s	4.01 s	0.13 s	0.017 s
% of total time	0.04%	0.63%	0.56%	1.15%	0.04%	0.01%

Table 7.9: Overhead of kMAF in % of the total execution time of each benchmark.

NAS-OMP										**PARSEC**												
BT-OMP	CG-OMP	DC-OMP	EP-OMP	FT-OMP	IS-OMP	LU-OMP	MG-OMP	SP-OMP	UA-OMP	Blackscholes	Bodytrack	Facesim	Ferret	Freqmine	Raytrace	Swaptions	Fluidanimate	Vips	X264	Canneal	Dedup	Streamcluster
1.8	1.8	2.0	0.8	2.6	1.6	1.9	1.4	2.4	1.9	1.3	1.5	0.8	1.2	1.0	1.4	1.6	1.6	1.0	1.4	1.9	1.2	2.3

all applications, the geometric mean of the overhead is 1.5%, which indicates a very low impact on the performance.

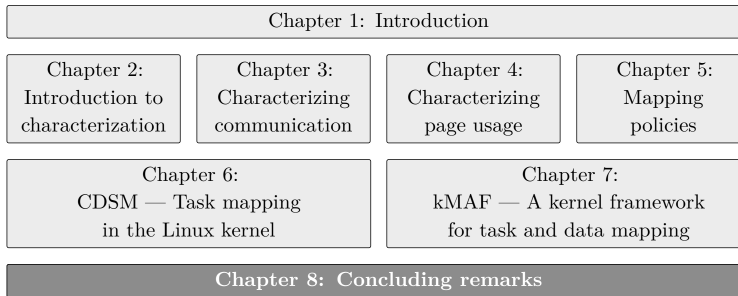
7.6 Summary

In this chapter, we introduced kMAF, which is a framework that automatically performs thread and data mapping on the operating system level. It requires no hardware changes and is compatible with all architectures that use virtual memory with paging. Furthermore, it requires no previous information about the applications' behavior, and

no changes to the applications themselves or their runtime libraries. kMAF itself is an extension of CDSM, which was presented in Chapter 6.

Experiments with a wide range of parallel applications with different memory access characteristics showed that kMAF was able to improve performance substantially on three different NUMA machines, with performance improvements of up to 300%. Energy consumption was also reduced, by up to 34.6% (9.3% on average). Results were close to an Oracle mechanism, and substantially better than previous work and simple mechanisms that do not take the application behavior into account. We also show that combining thread and data mapping can lead to higher improvements than performing them separately.

8 CONCLUDING REMARKS



Parallel architectures with complex memory access characteristics represent the state-of-the-art. These hierarchies are formed by multiple cache levels on the processors, as well as several main memories in NUMA machines. To fully benefit from these architectures, it is important to analyze the memory access characteristics of parallel applications and use this information to optimize the locality and balance of memory accesses. This can be done in two complementing ways: by running tasks that access shared data close to each other in the memory hierarchy (*task mapping*) and to place memory pages on the NUMA node that accesses them the most (*data mapping*). This thesis advances the field of mapping in two ways. First, we presented a technique to analyze parallel applications to determine their suitability for various types of mappings. Second, we introduced automatic mechanisms that perform the mapping automatically on the kernel level.

8.1 Summary of Main Results

In this thesis, we presented metrics and a methodology to analyze the memory access behavior of parallel applications to determine their potential for task and data mapping. We introduced CDSM, which is a kernel extension to detect communication of parallel applications and use this information to perform an optimized, communication-aware mapping of tasks to the processing units of the hardware architecture. We extended CDSM for kMAF, which is a framework that automatically performs data mapping in addition to the task mapping. Both mechanisms require no hardware changes and are compatible with all architectures that use virtual memory with paging. Furthermore, they require no previous information about the behavior of the parallel applications, and no changes to the applications themselves or their runtime libraries.

Experiments with a wide range of parallel applications with different memory access characteristics showed that the mechanisms were able to improve performance and energy consumption substantially on a variety of NUMA architectures. On our main evaluation system, the task mapping performed by CDSM improved performance by up to 21% (6% on average). On the same machine, kMAF was able to improve performance by up to 90% (14% on average). We also showed that task mapping is a prerequisite for data mapping. Combining both mappings can lead to higher improvements than performing both of them separately.

Results were close to an oracle mechanism, which has complete information about all memory accesses and has no runtime overhead, and substantially better than simpler mechanisms that do not take the actual application behavior into account. Since CDSM

is performing only task mapping, it is simpler to implement than kMAF. Furthermore, its runtime overhead is much smaller than kMAF's, both on the execution time and memory consumption, since it needs to create more extra page faults and needs to store the usage pattern of each page. Nevertheless, kMAF only caused an execution time overhead of less than 3% for all benchmarks, compared to less than 1% with CDSM.

8.2 Released Software

Several pieces of software were developed as part of this thesis, which were described in detail in the previous chapters. This section contains a brief summary of this software. All software is released as open source, licensed under the GPL-v2 license.

1. **Numalize.** Numalize is a trace-based mechanism to analyze the communication and page usage of parallel applications, and can calculate optimized task and data mappings with various mapping policies. Numalize was used for the characterizations in Chapters 3 and 4, and for the Oracle mapping mechanism described in Section 5.4 and evaluated in Sections 6.5 and 7.5. It is available at <https://github.com/matthiasdiener/numalize>.
2. **CDSM.** CDSM is a module for the Linux kernel to perform automatic task mapping and is discussed in Chapter 6. CDSM is available at <https://github.com/matthiasdiener/cdsm>.
3. **kMAF.** kMAF is a module for the Linux kernel for automatic task and data mapping, discussed in Chapter 7. It is available at <https://github.com/matthiasdiener/kmaf>.

8.3 Research Perspectives

Several opportunities were identified to extend the research that was presented in this thesis.

1. **Task and data mapping in embedded systems.** This thesis focuses on performing mapping in large traditional computer architectures with HPC characteristics, which shows high improvements due to their high parallelism. However, the parallelism in embedded system, such as in Multi-Processor System-on-Chip (MPSoC) is also increasing, which leads to opportunities for task mapping in these architectures. Many current MPSoC systems, such as Infineon's Aurix platform (INFINEON, 2014) also contain a Non-Uniform Memory Access Behavior (PAPAGIANNOPOULOU et al., 2013), which can make data mapping important as well. Perform mapping on these architectures has some specific issues, such as restrictions on the latency of applications, that can increase the challenges.
2. **Hybrid mapping mechanisms.** The automatic mechanisms presented in this thesis have a limitation compared to other mechanisms such as the Oracle or a manual mapping in that they have no prior information about application behavior and need to detect it during execution. Although CDSM and kMAF showed gains that were very close to the mechanisms with prior information for all applications that were evaluated, this is not necessarily true for all possible application behaviors. For example, programs with a

very short execution time or a very dynamic behavior might benefit substantially from prior information. However, current solutions that perform mapping with prior behavior execute in user-space, which severely limits their applicability in general-purpose systems. Multiple running applications can not be supported in user-space for instance, and mapping decisions would be interfering with each other.

For this reason, a *hybrid* mechanism can be an interesting solution to combine the prior information generated in user-space with the mapping decisions performed by the kernel during runtime. In this way, the analysis of the memory access behavior can be performed before execution by the compiler, the developer or a trace-based mechanism. This information can then be passed to the operating system kernel, which uses the analyzed behavior to perform mapping decisions. The kernel can refine the mapping during execution, due to changes external to the application (such as starting another application, memory is added or removed from the system, among others), or internal changes (such as when the provided memory access behavior is incorrect). For such a refinement of the mapping, mechanisms such as kMAF can be used.

3. Task mapping in manycore architectures. As mentioned in Section 1.1.2, manycore architectures with thousands of cores create interesting challenges, particularly for task mapping. It will not be possible to execute a *global* task mapping algorithm, such as the one used in this thesis, to optimize the communication behavior, due to the large runtime overhead. Two solutions can be used to overcome this issue. First, by partitioning the manycore processor into several blocks and then applying such an algorithm to each part can reduce the overhead of the task mapping. In case an application does not use the full number of cores, this can be a viable solution. Second, by performing the mapping *locally*, that is, for only a small subset of tasks, no global mapping needs to be calculated. For example, when the kernel notices that two tasks are communicating a lot, it can migrate one task closer to the other, without affecting the other tasks. This resembles the spirit of our data mapping policy, which migrates single pages without calculating a global access behavior.

4. Heterogeneous architectures. In this thesis, we assumed that architectures are *homogeneous*, that is, PUs have the same number of functional units, processors have a symmetrical cache hierarchy and there is only a simple local/remote hierarchy of memory controllers. However, modern systems are starting to have a *heterogeneous* behavior. For example, processors such as ARM's big.LITTLE have some cores that have a higher computational power and energy consumption than other cores on the same chip. A solution could be to add this information to the task mapping algorithm. Furthermore, architectures with multiple memory controllers on the same chip, such as the *Opteron* machine used in this thesis, include a hierarchical NUMA behavior, with memory accesses to the local NUMA node, a remote node on the same chip and a remote node on a different chip. This hierarchy issue was not explored in this thesis, but can be added to the data mapping policies, for example by reducing data migrations between NUMA nodes on the same chip.

5. Integrate mechanisms deeper into kernel. Even though CDSM and kMAF are integrated into the Linux kernel, it is interesting to integrate them more tightly with

the kernel itself to improve the efficiency of operations and remove duplication of code and data structures. For task mapping, our mapping algorithm overrides the normal scheduler of Linux and only takes communication into account to generate a mapping. An alternative could be to provide information generated from the communication detection (such as the communication matrix) to the task scheduler, which can then calculate an optimized mapping that takes into account the communication, load balance, and other factors. For data mapping, where we store information about every page, information about the memory access pattern to the pages could be stored directly in the `struct page` structure instead of a separate hash table. In this way, the memory usage and complexity of data mapping can be reduced significantly.

6. Hardware-assisted application analysis. The analysis of our mapping mechanisms is based on the observation and introduction of page faults during execution. Such a mechanism has the advantage of being easily portable between different hardware architectures, as virtual memory management with demand paging is the most common technique in modern systems. However, using page faults to detect memory accesses has some drawbacks, which make mapping mechanisms have a higher complexity and overhead. For example, introducing additional page faults creates an overhead for the running application. Furthermore, great care has to be taken to distinguish these extra faults from faults that indicate that a page is located in swap, further complicating the implementation of such a mechanism.

These disadvantages could be solved by a dedicated mechanism that provides sampled memory addresses to the operating system. The operating system could then use these addresses instead of page faults to characterize communication and page usage of the applications. Several previous mechanisms discussed in this thesis (such as Marathe et al. (2006)) already use PMU statistics for mapping, but suffer from a substantial overhead. A more modern replacement could be based on proposals such as AMD's Lightweight Profiling (LWP) (AMD, 2010), which allows sampling of memory accesses with full addresses and a configurable sampling granularity.

8.4 Publications

The following papers (listed in reverse chronological order) were published since entering the PhD program and contain material that is relevant to this thesis:

1. Matthias Diener, Eduardo H. M. Cruz, Marco A. Z. Alves, Mohammad S. Alhakeem, Philippe O. A. Navaux, Hans-Ulrich Heiß. **"Locality and Balance for Communication-Aware Thread Mapping in Multicore Systems."** Euro-Par, August 2015.

2. Matthias Diener, Eduardo H. M. Cruz, Laércio L. Pilla, Fabrice Dupros, Philippe O. A. Navaux. **"Characterizing Communication and Page Usage of Parallel Applications for Thread and Data Mapping."** Performance Evaluation, June 2015.

3. Eduardo H. M. Cruz, Matthias Diener, Philippe O. A. Navaux. **"Communication-Aware Thread Mapping Using the Translation Lookaside Buffer."** Concurrency and Computation: Practice and Experience, 2015.

4. Matthias Diener, Eduardo H. M. Cruz, Philippe O. A. Navaux, Anselm Busse, Hans-Ulrich Heiß. **"Communication-Aware Process and Thread Mapping Using Online Communication Detection."** Journal of Parallel Computing (PARCO), March 2015.

5. Matthias Diener, Eduardo H. M. Cruz, Philippe O. A. Navaux. **"Locality vs. Balance: Exploring Data Mapping Policies on NUMA Systems."** Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), March 2015.

6. Eduardo H. M. Cruz, Matthias Diener, Laércio L. Pilla, Philippe O. A. Navaux. **"An Efficient Algorithm for Communication-Based Task Mapping."** Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), March 2015. (**Best Paper Award**)

7. Eduardo H. M. Cruz, Matthias Diener, Marco A. Z. Alves, Laércio L. Pilla, Philippe O. A. Navaux. **"Optimizing Memory Locality Using a Locality-Aware Page Table."** International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), October 2014.

8. Matthias Diener, Eduardo H. M. Cruz, Philippe O. A. Navaux, Anselm Busse, Hans-Ulrich Heiß. **"kMAF: Automatic Kernel-Level Management of Thread and Data Affinity."** International Conference on Parallel Architectures and Compilation Techniques (PACT), August 2014.

9. Eduardo H. M. Cruz, Matthias Diener, Marco A. Z. Alves, Philippe O. A. Navaux. **"Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols."** Journal of Parallel and Distributed Computing (JPDC), March 2014.

10. Matthias Diener, Eduardo H. M. Cruz, Philippe O. A. Navaux. **"Communication-Based Mapping using Shared Pages."** International Parallel & Distributed Processing Symposium (IPDPS), May 2013.

11. Eduardo H. M. Cruz, Matthias Diener, Philippe O. A. Navaux. **"Using the Translation Lookaside Buffer to Map Threads in Parallel Applications Based on Shared Memory."** International Parallel & Distributed Processing Symposium (IPDPS), May 2012.

REFERENCES

ADHIANTO, L. et al. HPCToolkit: Tools for performance analysis of optimized parallel programs. **Concurrency Computation Practice and Experience**, v. 22, n. 6, p. 685–701, 2010.

AGARWAL, V. et al. Clock rate versus IPC: The end of the road for conventional microarchitectures. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE (ISCA). **Proceedings...** Vancouver: IEEE, 2000. p. 248–259.

ALVES, M. A. Z. **Increasing Energy Efficiency of Processor Caches via Line Usage Predictors**. Thesis (PhD) — Federal University of Rio Grande do Sul, 2014.

ALVES, M. A. Z. et al. SiNUCA: A Validated Micro-Architecture Simulator. In: IEEE INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE COMPUTING AND COMMUNICATIONS (HPCC). **Proceedings...** [S.l.], 2015.

AMD. **AMD64 Technology – Lightweight Profiling Specification**. [S.l.], 2010.

AMD. **AMD Opteron™ 6300 Series processor Quick Reference Guide**. [S.l.], 2012.

ANTONY, J.; JANES, P.; RENDELL, A. Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. In: HIGH PERFORMANCE COMPUTING (HIPC). **Proceedings...** [S.l.], 2006. p. 338–352.

AOCHI, H. et al. Finite difference simulations of seismic wave propagation for understanding earthquake physics and predicting ground motions: Advances and challenges. **Journal of Physics: Conference Series**, v. 454, n. 1, p. 012010, aug 2013.

ARCANGELI, A. Transparent Hugepage Support. In: KVM FORUM. **Proceedings...** [S.l.], 2010.

ARGONNE NATIONAL LABORATORY. **CH3 And Channels**. 2014. Available from Internet: <https://wiki.mpich.org/mpich/index.php/CH3_And_Channels>.

ARGONNE NATIONAL LABORATORY. **Using the Hydra Process Manager**. 2014. Available from Internet: <http://wiki.mpich.org/mpich/index.php/Using_the_Hydra_Process_Manager>.

ASANOVIC, K. et al. **The Landscape of Parallel Computing Research: A View from Berkeley**. [S.l.], 2006. 56 p.

AWASTHI, M. et al. Handling the Problems and Opportunities Posed by Multiple On-Chip Memory Controllers. In: PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES (PACT). **Proceedings...** [S.l.], 2010. p. 319–330.

AZIMI, R. et al. Enhancing Operating System Support for Multicore Processors by Using Hardware Performance Monitoring. **ACM SIGOPS Operating Systems Review**, v. 43, n. 2, p. 56–65, apr 2009.

BACH, M. et al. Analyzing Parallel Programs with Pin. **IEEE Computer**, IEEE, v. 43, n. 3, p. 34–41, 2010.

BAILEY, D. et al. **The NAS parallel benchmarks 2.0**. [S.l.], 1995. 1–24 p. Available from Internet: <http://www.nas.nasa.gov/assets/pdf/techreports/1995/nas-95-020.pdf>.

BAILEY, D. H. et al. The NAS Parallel Benchmarks. **International Journal of High Performance Computing Applications**, v. 5, n. 3, p. 66–73, 1991.

BALLARD, G. et al. Communication lower bounds and optimal algorithms for numerical linear algebra. **Acta Numerica**, v. 23, n. May, p. 1–155, 2014.

BARROW-WILLIAMS, N.; FENSCH, C.; MOORE, S. A Communication Characterisation of Splash-2 and Parsec. In: IEEE INTERNATIONAL SYMPOSIUM ON WORKLOAD CHARACTERIZATION (IISWC). **Proceedings...** [S.l.], 2009. p. 86–97.

BASU, A. et al. Efficient Virtual Memory for Big Memory Servers. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE (ISCA). **Proceedings...** [S.l.], 2013. p. 237–248.

BELLOSA, F.; STECKERMEIER, M. The performance implications of locality information usage in shared-memory multiprocessors. **Journal of Parallel and Distributed Computing**, v. 37, n. 1, p. 113–121, 1996.

BIENIA, C.; KUMAR, S.; LI, K. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on Chip-Multiprocessors. In: IEEE INTERNATIONAL SYMPOSIUM ON WORKLOAD CHARACTERIZATION (IISWC). **Proceedings...** [S.l.], 2008. p. 47–56.

BIENIA, C. et al. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES (PACT). **Proceedings...** [S.l.], 2008. p. 72–81.

BINKERT, N. et al. The Gem5 Simulator. **SIGARCH Computer Architecture News**, v. 39, n. 2, p. 1–7, 2011.

BLAGODUROV, S. et al. A Case for NUMA-aware Contention Management on Multicore Systems. In: USENIX ANNUAL TECHNICAL CONFERENCE (ATC). **Proceedings...** [S.l.], 2010. p. 557–571.

BLUMOFE, R. D.; LEISERSON, C. E. Scheduling multithreaded computations by work stealing. In: SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE (FOCS). **Proceedings...** [S.l.], 1994. p. 1–29.

BOKHARI, S. On the Mapping Problem. **IEEE Transactions on Computers**, C-30, n. 3, p. 207–214, 1981.

BOLOSKY, W. J.; SCOTT, M. L. Evaluation of multiprocessor memory systems using off-line optimal behavior. **Journal of Parallel and Distributed Computing (JPDC)**, v. 15, n. 4, p. 382–398, 1992.

BOLOSKY, W. J.; SCOTT, M. L. False sharing and its effect on shared memory performance. **USENIX Systems on USENIX Experiences**, USENIX Association, v. 1801, n. 14520052, p. 1–15, 1993.

BRANDFASS, B.; ALRUTZ, T.; GERHOLD, T. Rank reordering for MPI communication optimization. **Computers & Fluids**, Elsevier Ltd, p. 372–380, jan 2012.

BROQUEDIS, F. et al. Structuring the execution of OpenMP applications for multicore architectures. In: IEEE INTERNATIONAL PARALLEL & DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS). **Proceedings...** [S.l.], 2010. p. 1–10.

BROQUEDIS, F. et al. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In: EUROMICRO CONFERENCE ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING (PDP). **Proceedings...** [S.l.], 2010. p. 180–186.

BUNTINAS, D. et al. Cache-Efficient, Intranode, Large-Message MPI Communication with MPICH2-Nemesis. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING (ICPP). **Proceedings...** [S.l.], 2009. p. 462–469.

BUNTINAS, D.; MERCIER, G.; GROPP, W. Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem. In: INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID (CCGRID). **Proceedings...** [S.l.], 2006. p. 521–530.

BUNTINAS, D.; MERCIER, G.; GROPP, W. Implementation and shared-memory evaluation of MPICH2 over the Nemesis communication subsystem. In: RECENT ADVANCES IN PARALLEL VIRTUAL MACHINE AND MESSAGE PASSING INTERFACE. **Proceedings...** [S.l.], 2006.

BUTTLAR, D.; FARRELL, J. **Pthreads programming: A POSIX standard for better multiprocessing**. [S.l.: s.n.], 1996.

CASCAVAL, C. et al. Multiple page size modeling and optimization. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES (PACT). **Proceedings...** [S.l.], 2005. p. 339–349.

CHAN, A.; GROPP, W.; LUSK, E. **User's Guide for MPE Extensions for MPI Programs**. [S.l.], 1998.

CHEN, H. et al. MPIPP: An Automatic Profile-guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters. In: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING (SC). **Proceedings...** [S.l.], 2006. p. 353–360.

CHISHTI, Z.; POWELL, M. D.; VIJAYKUMAR, T. N. Optimizing Replication, Communication, and Capacity Allocation in CMPs. **ACM SIGARCH Computer Architecture News**, v. 33, n. 2, p. 357–368, may 2005.

CHODNEKAR, S. et al. Towards a communication characterization methodology for parallel applications. In: INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE (HPCA). **Proceedings...** [S.l.], 1997. p. 310–319.

CONWAY, P. The AMD Opteron Northbridge Architecture. **IEEE Micro**, v. 27, n. 2, p. 10–21, 2007.

CORBET, J. **AutoNUMA: the other approach to NUMA scheduling**. 2012. Available from Internet: <http://lwn.net/Articles/488709/>.

CORBET, J. **Toward better NUMA scheduling**. 2012. Available from Internet: <http://lwn.net/Articles/486858/>.

COTEUS, P. W. et al. Technologies for exascale systems. **IBM Journal of Research and Development**, v. 55, n. 5, p. 14:1–14:12, sep 2011.

CRUZ, E. H. M. et al. Using Memory Access Traces to Map Threads and Data on Hierarchical Multi-core Platforms. In: IEEE INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING WORKSHOPS AND PHD FORUM. **Proceedings...** [S.l.], 2011. p. 551–558.

CRUZ, E. H. M. et al. Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols. **Journal of Parallel and Distributed Computing (JPDC)**, v. 74, n. 3, p. 2215–2228, mar 2014.

CRUZ, E. H. M. et al. Optimizing Memory Locality Using a Locality-Aware Page Table. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING (SBAC-PAD). **Proceedings...** [S.l.], 2014. p. 198–205.

CRUZ, E. H. M.; DIENER, M.; NAVAUX, P. O. A. Using the Translation Lookaside Buffer to Map Threads in Parallel Applications Based on Shared Memory. In: IEEE INTERNATIONAL PARALLEL & DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS). **Proceedings...** [S.l.], 2012. p. 532–543.

CRUZ, E. H. M.; DIENER, M.; NAVAUX, P. O. A. Communication-Aware Thread Mapping Using the Translation Lookaside Buffer. **Concurrency Computation: Practice and Experience**, v. 22, n. 6, p. 685–701, 2015.

CRUZ, E. H. M. et al. An Efficient Algorithm for Communication-Based Task Mapping. In: INTERNATIONAL CONFERENCE ON PARALLEL, DISTRIBUTED, AND NETWORK-BASED PROCESSING (PDP). **Proceedings...** [S.l.], 2015. p. 207–214.

DAGUM, L.; MENON, R. OpenMP: An Industry-Standard API for Shared-Memory Programming. **IEEE Computational Science and Engineering (CSE)**, v. 5, n. 1, p. 46–55, 1998.

DALLY, W. J. **GPU Computing to Exascale and Beyond**. [S.l.], 2010.

DASHTI, M. et al. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In: ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS (ASPLOS). **Proceedings...** [S.l.], 2013. p. 381–393.

DEROSE, L. et al. SIGMA: A Simulator Infrastructure to Guide Memory Analysis. In: INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS (SC). **Proceedings...** [S.l.], 2002. p. 1–13.

DEVINE, K. D. et al. Parallel hypergraph partitioning for scientific computing. In: IEEE INTERNATIONAL PARALLEL & DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS). **Proceedings...** [S.l.], 2006. p. 124–133.

DIENER, M. **Evaluating Thread Placement Improvements in Multi-core Architectures**. Dissertation (Master) — Berlin Institute of Technology, 2010.

DIENER, M.; CRUZ, E. H. M.; NAVAUX, P. O. A. Communication-Based Mapping Using Shared Pages. In: IEEE INTERNATIONAL PARALLEL & DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS). **Proceedings...** [S.l.], 2013. p. 700–711.

DIENER, M.; CRUZ, E. H. M.; NAVAUX, P. O. A. Locality vs . Balance: Exploring Data Mapping Policies on NUMA Systems. In: INTERNATIONAL CONFERENCE ON PARALLEL, DISTRIBUTED, AND NETWORK-BASED PROCESSING (PDP). **Proceedings...** [S.l.], 2015. p. 9–16.

DIENER, M. et al. kMAF: Automatic Kernel-Level Management of Thread and Data Affinity. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES (PACT). **Proceedings...** [S.l.], 2014. p. 277–288.

DIENER, M. et al. Characterizing Communication and Page Usage of Parallel Applications for Thread and Data Mapping. **Performance Evaluation**, v. 88-89, n. June, p. 18–36, 2015.

DIENER, M. et al. Evaluating Thread Placement Based on Memory Access Patterns for Multi-core Processors. In: IEEE INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE COMPUTING AND COMMUNICATIONS (HPCC). **Proceedings...** [S.l.], 2010. p. 491–496.

DREPPER, U. **What Every Programmer Should Know About Memory**. [S.l.], 2007. v. 3, n. 4, 114 p. Available from Internet: <http://www.akkadia.org/drepper/cpumemory.pdf>.

DREPPER, U.; MOLNAR, I. **The Native POSIX Thread Library for Linux**. [S.l.], 2002.

DRONGOWSKI, P. J. **Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors**. [S.l.], 2007.

DUPROS, F. et al. Exploiting Intensive Multithreading for the Efficient Simulation of 3D Seismic Wave Propagation. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTATIONAL SCIENCE AND ENGINEERING (CSE). **Proceedings...** [S.l.], 2008. p. 253–260.

DUPROS, F. et al. Parallel simulations of seismic wave propagation on NUMA architectures. In: PARALLEL COMPUTING: FROM MULTICORES AND GPU'S TO PETASCALE. **Proceedings...** [S.l.], 2010. p. 67–74.

EKMAN, P. et al. **Design Considerations for Shared Memory MPI Implementations on Linux NUMA Systems: An MPICH/MPICH2 Case Study.** [S.l.], 2005.

FARAJ, A.; YUAN, X. Communication characteristics in the NAS parallel benchmarks. In: PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS (PDCS). **Proceedings...** [S.l.], 2002. p. 724–729.

FELIU, J. et al. Understanding Cache Hierarchy Contention in CMPs to Improve Job Scheduling. In: INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS). **Proceedings...** [S.l.], 2012.

FREITAS, S. R. et al. The Coupled Aerosol and Tracer Transport model to the Brazilian developments on the Regional Atmospheric Modeling System (CATT-BRAMS) – Part 1: Model description and evaluation. **Atmospheric Chemistry and Physics**, v. 9, n. 8, p. 2843–2861, apr 2009.

FRIEDLEY, A. et al. Hybrid MPI: Efficient Message Passing for Multi-core Systems. In: INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS (SC). **Proceedings...** [S.l.], 2013. p. 1–11.

GABRIEL, E. et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In: RECENT ADVANCES IN PARALLEL VIRTUAL MACHINE AND MESSAGE PASSING INTERFACE. **Proceedings...** [S.l.], 2004.

GAUD, F. et al. Large Pages May Be Harmful on NUMA Systems. In: USENIX ANNUAL TECHNICAL CONFERENCE (ATC). **Proceedings...** [S.l.], 2014. p. 231–242.

GIMÉNEZ, A. et al. Dissecting On-Node Memory Access Performance: A Semantic Approach. In: INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS (SC). **Proceedings...** [S.l.], 2014. p. 166–176.

GLANTZ, R.; MEYERHENKE, H.; NOE, A. Algorithms for Mapping Parallel Processes onto Grid and Torus Architectures. In: INTERNATIONAL CONFERENCE ON PARALLEL, DISTRIBUTED, AND NETWORK-BASED PROCESSING (PDP). **Proceedings...** [S.l.], 2015. p. 236–243.

GOGLIN, B.; FURMENTO, N. Enabling High-Performance Memory Migration for Multithreaded Applications on Linux. In: IEEE INTERNATIONAL SYMPOSIUM ON PARALLEL & DISTRIBUTED PROCESSING (IPDPS). **Proceedings...** [S.l.], 2009. p. 1–9.

GOGLIN, B.; MOREAUD, S. KNEM: A generic and scalable kernel-assisted intra-node MPI communication framework. **Journal of Parallel and Distributed Computing**, Elsevier Inc., v. 73, n. 2, p. 176–188, feb 2013.

GROPP, W. MPICH2: A new start for MPI implementations. In: RECENT ADVANCES IN PARALLEL VIRTUAL MACHINE AND MESSAGE PASSING INTERFACE. **Proceedings...** [S.l.], 2002.

HORE, A.; ZIOU, D. Image Quality Metrics: PSNR vs. SSIM. In: INTERNATIONAL CONFERENCE ON PATTERN RECOGNITION. **Proceedings...** [S.l.], 2010. p. 2366–2369.

HURSEY, J.; SQUYRES, J.; DONTJE, T. Locality-Aware Parallel Process Mapping for Multi-Core HPC Systems. In: IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING (CLUSTER). **Proceedings...** [S.l.], 2011. p. 527–531.

INFINEON. **AURIX TC29x B-Step: User's Manual**. [S.l.], 2014.

INTEL. **Dual-Core Intel® Itanium® Processor 9000 and 9100 Series**. [S.l.], 2007.

INTEL. **Quad-Core Intel ® Xeon ® Processor 5400 Series Datasheet**. [S.l.], 2008. Available from Internet: <http://www.intel.com/assets/PDF/datasheet/318589.pdf>.

INTEL. **Intel® Itanium® Architecture Software Developer's Manual**. [S.l.], 2010.

INTEL. **Intel® Xeon® Processor 7500 Series**. [S.l.], 2010.

INTEL. **Intel Performance Counter Monitor - A better way to measure CPU utilization**. 2012. Available from Internet: <http://www.intel.com/software/pcm>.

INTEL. **Using KMP_AFFINITY to create OpenMP thread mapping to OS proc IDs**. 2012. Available from Internet: <https://software.intel.com/en-us/articles/openmp-thread-affinity-control>.

INTEL. **Intel Trace Analyzer and Collector**. 2013. Available from Internet: <http://software.intel.com/en-us/intel-trace-analyzer>.

INTEL. **Intel© 64 and IA-32 Architectures Software Developer's Manual**. 2013.

ITO, S.; GOTO, K.; ONO, K. Automatically optimized core mapping to subdomains of domain decomposition method on multicore parallel environments. **Computers & Fluids**, Elsevier Ltd, v. 80, p. 88–93, jul 2013.

JEANNOT, E.; MERCIER, G. Near-optimal placement of MPI processes on hierarchical NUMA architectures. In: EURO-PAR PARALLEL PROCESSING. **Proceedings...** [S.l.], 2010. p. 199–210.

JEANNOT, E.; MERCIER, G.; TESSIER, F. Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques. **IEEE Transactions on Parallel and Distributed Systems**, v. 25, n. 4, p. 993–1002, apr 2014.

JIN, H.; FRUMKIN, M.; YAN, J. **The OpenMP implementation of NAS Parallel Benchmarks and Its Performance**. [S.l.], 1999.

KALE, L. V.; KRISHNAN, S. CHARM++: A Portable Concurrent Object Oriented System Based On C++. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS (OOPSLA). **Proceedings...** [S.l.], 1993. p. 91–108.

KARYPIS, G.; KUMAR, V. Parallel Multilevel Graph Partitioning. In: INTERNATIONAL PARALLEL PROCESSING SYMPOSIUM (IPPS). **Proceedings...** [S.l.], 1996. p. 314–319.

KARYPIS, G.; KUMAR, V. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. **SIAM Journal on Scientific Computing**, v. 20, n. 1, p. 359–392, jan 1998.

KEJARIWAL, A. et al. Comparative Architectural Characterization of SPEC CPU2000 and CPU2006 Benchmarks on the Intel Core 2 Duo Processor. In: INTERNATIONAL CONFERENCE ON EMBEDDED COMPUTER SYSTEMS: ARCHITECTURES, MODELING AND SIMULATION (SAMOS). **Proceedings...** [S.l.], 2008. p. 132–141.

KIM, J.; LILJA, D. J. Characterization of communication patterns in message-passing parallel scientific application programs. In: INTERNATIONAL WORKSHOP ON NETWORK-BASED PARALLEL COMPUTING: COMMUNICATION, ARCHITECTURE, AND APPLICATIONS (CANPC). **Proceedings...** [S.l.], 1998. p. 202–216.

KLEEN, A. **An NUMA API for Linux**. 2004. Available from Internet: <http://andikleen.de/numaapi3.pdf>.

KLUG, T. et al. autopin – Automated Optimization of Thread-to-Core Pinning on Multicore Systems. **High Performance Embedded Architectures and Compilers (HiPEAC)**, v. 3, n. 4, p. 219–235, 2008.

LACHAIZE, R.; LEPERS, B.; QUÉMA, V. MemProf: A Memory Profiler for NUMA Multicore Systems. In: USENIX ANNUAL TECHNICAL CONFERENCE (ATC). **Proceedings...** [S.l.], 2012. p. 53–64.

LANKES, S.; BIERBAUM, B.; BEMMERL, T. Affinity-On-Next-Touch: An Extension to the Linux Kernel for NUMA Architectures. **Lecture Notes in Computer Science**, v. 6067 LNCS, n. PART 1, p. 576–585, 2010.

LAROWE JR, R. P. **Page Placement For Non-Uniform Memory Access Time (NUMA) Shared Memory Multiprocessors**. Thesis (PhD) — Duke University, 1991.

LAROWE, R. P.; HOLLIDAY, M. A.; ELLIS, C. S. An Analysis of Dynamic Page Placement on a NUMA Multiprocessor. **ACM SIGMETRICS Performance Evaluation Review**, v. 20, n. 1, p. 23–34, 1992.

LEE, I. Characterizing communication patterns of NAS-MPI benchmark programs. In: IEEE SOUTHEASTCON. **Proceedings...** [S.l.], 2009. p. 158–163.

LEVINTHAL, D. **Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors**. [S.l.], 2009.

LI, S. et al. The McPAT Framework for Multicore and Manycore Architectures: Simultaneously Modeling Power, Area, and Timing. **ACM Transactions on Architecture and Code Optimization**, v. 10, n. 1, p. 1–29, 2013.

LIU, X.; MELLOR-CRUMMEY, J. A tool to Analyze the Performance of Multithreaded Programs on NUMA Architectures. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING (PPOPP). **Proceedings...** [S.l.], 2014. p. 259–272.

LÖF, H.; HOLMGREN, S. affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System. In: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING (SC). **Proceedings...** [S.l.], 2005. p. 387–392.

LONG, D. L.; CLARKE, L. A. Task Interaction Graphs for Concurrency Analysis. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING. **Proceedings...** [S.l.], 1989. p. 44–52.

LU, H. J. et al. Using Hugetlbfs for Mapping Application Text Regions. In: LINUX SYMPOSIUM. **Proceedings...** [S.l.], 2006. p. 75–82.

LUK, C. et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION (PLDI). **Proceedings...** [S.l.], 2005. p. 190–200.

LUSZCZEK, P. et al. **Introduction to the HPC Challenge Benchmark Suite**. [S.l.], 2005.

MA, C. et al. An Approach for Matching Communication Patterns in Parallel Applications. In: IEEE INTERNATIONAL PARALLEL & DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS). **Proceedings...** [S.l.], 2009. p. 1–12.

MAGNUSSON, P. et al. Simics: A Full System Simulation Platform. **IEEE Computer**, IEEE Computer Society, v. 35, n. 2, p. 50–58, 2002.

MAJO, Z.; GROSS, T. R. Matching memory access patterns and data placement for NUMA systems. In: INTERNATIONAL SYMPOSIUM ON CODE GENERATION AND OPTIMIZATION (CGO). **Proceedings...** [S.l.], 2012. p. 230–241.

MAJO, Z.; GROSS, T. R. (Mis)Understanding the NUMA Memory System Performance of Multithreaded Workloads. In: IEEE INTERNATIONAL SYMPOSIUM ON WORKLOAD CHARACTERIZATION (IISWC). **Proceedings...** [S.l.], 2013. p. 11–22.

MARATHE, J.; MUELLER, F. Hardware Profile-guided Automatic Page Placement for ccNUMA Systems. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING (PPOPP). **Proceedings...** [S.l.], 2006. p. 90–99.

MARATHE, J.; THAKKAR, V.; MUELLER, F. Feedback-Directed Page Placement for ccNUMA via Hardware-generated Memory Traces. **Journal of Parallel and Distributed Computing (JPDC)**, v. 70, n. 12, p. 1204–1219, 2010.

MARCHETTI, M. et al. Using Simple Page Placement Policies to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems. In: INTERNATIONAL PARALLEL PROCESSING SYMPOSIUM (IPPS). **Proceedings...** [S.l.], 1995. p. 480–485.

MARCO-GISBERT, H.; RIPOLL, I. **On the Effectiveness of Full-ASLR on 64-bit Linux**. [S.l.], 2014.

MARIANO, A.; LAARHOVEN, T.; BISCHOF, C. Parallel (probable) lock-free HashSieve: a practical sieving algorithm for the SVP. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING (ICPP). **Proceedings...** [S.l.], 2015.

MATTSON, T.; MEADOWS, L. **A "Hands-on" Introduction to OpenMP**. 2008.

MAVINAKAYANAHALLI, A. et al. Probing the Guts of Kprobes. In: LINUX SYMPOSIUM. **Proceedings...** [S.l.], 2006. p. 101–116.

MCCALPIN, J. D. **Sustainable Memory Bandwidth in Current High Performance Computers**. [S.l.], 1995. Available from Internet: <http://www.cs.virginia.edu/~mccalpin/papers/bandwidth/bandwidth.html>.

MCCURDY, C.; VETTER, J. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In: IEEE INTERNATIONAL SYMPOSIUM ON PERFORMANCE ANALYSIS OF SYSTEMS & SOFTWARE (ISPASS). **Proceedings...** [S.l.], 2010. p. 87–96.

MCDOUGALL, R. **Supporting Multiple Page Sizes in the Solaris™ Operating System**. [S.l.], 2004.

MCVOY, L.; STAELIN, C. Lmbench: Portable Tools for Performance Analysis. In: USENIX ANNUAL TECHNICAL CONFERENCE (ATC). **Proceedings...** [S.l.], 1996. p. 23–38.

MELO, A. C. de. The New Linux 'perf' Tools. In: LINUX KONGRESS. **Proceedings...** [S.l.], 2010.

MERCIER, G.; CLET-ORTEGA, J. Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments. In: RECENT ADVANCES IN PARALLEL VIRTUAL MACHINE AND MESSAGE PASSING INTERFACE. **Proceedings...** [S.l.], 2009. p. 104–115.

MERCIER, G.; JEANNOT, E. Improving MPI Applications Performance on Multicore Clusters with Rank Reordering. In: EUROPEAN MPI USERS' GROUP CONFERENCE ON RECENT ADVANCES IN THE MESSAGE PASSING INTERFACE (EUROMPI). **Proceedings...** [S.l.], 2011.

MESSAGE PASSING INTERFACE FORUM. **MPI: A Message-Passing Interface Standard**. [S.l.], 2012.

METER, R. V. **Memory: Caching and Memory Hierarchy**. 2009. Available from Internet: <http://web.sfc.keio.ac.jp/~rdv/keio/sfc/teaching/architecture/architecture-2009/lec08-cache.html>.

MICCIANCIO, D. Shortest vector problem. In: COMPLEXITY OF LATTICE PROBLEMS. **Proceedings...** [S.l.], 2002. p. 69–90.

NAVARRO, J. E. **Transparent operating system support for superpages**. Thesis (PhD) — Rice University, 2004.

NETHERCOTE, N. **Dynamic Binary Analysis and Instrumentation or Building Tools is Easy**. Thesis (PhD) — University of Cambridge, 2004.

NETHERCOTE, N.; SEWARD, J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. **ACM Sigplan Notices**, v. 42, n. 6, p. 89–100, 2007.

NIKOLOPOULOS, D. S. et al. UPMLIB: A runtime system for Turning the memory performance of OpenMP programs on scalable shared-memory multiprocessors. In: LANGUAGES, COMPILERS, AND RUN-TIME SYSTEMS FOR SCALABLE COMPUTERS. **Proceedings...** [S.l.], 2000. p. 85–99.

NIKOLOPOULOS, D. S. et al. User-level dynamic page migration for multiprogrammed shared-memory multiprocessors. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING (ICPP). **Proceedings...** [S.l.], 2000. p. 95–103.

NITZBERG, B.; LO, V. Distributed Shared Memory: A Survey of Issues and Algorithms. **IEEE Computer**, v. 24, n. 8, p. 52–60, 1991.

OGASAWARA, T. NUMA-Aware Memory Manager with Dominant-Thread-Based Copying GC. **ACM SIGPLAN Notices**, v. 44, n. 10, p. 377–389, oct 2009.

ORACLE. **Solaris OS Tuning Features**. 2010. Available from Internet: <http://docs.oracle.com/cd/E18659_01/html/821-1381/aewda.html>.

PAAS, R. van der. **Getting OpenMP Up To Speed**. [S.l.], 2009.

PAPAGIANNOPOULOU, D. et al. Transparent and energy-efficient speculation on NUMA architectures for embedded MPSoCs. In: INTERNATIONAL WORKSHOP ON MANY-CORE EMBEDDED SYSTEMS (MES). **Proceedings...** [S.l.], 2013. p. 58–61.

PAYER, M.; KRAVINA, E.; GROSS, T. R. Lightweight Memory Tracing. In: USENIX ANNUAL TECHNICAL CONFERENCE (ATC). **Proceedings...** [S.l.], 2013. p. 115–126.

PEARCE, O. et al. Quantifying the effectiveness of load balance algorithms. In: ACM INTERNATIONAL CONFERENCE ON SUPERCOMPUTING (ICS). **Proceedings...** [S.l.], 2012. p. 185–194.

PELLEGRINI, F. Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In: SCALABLE HIGH-PERFORMANCE COMPUTING CONFERENCE (SHPCC). **Proceedings...** [S.l.], 1994. p. 486–493.

PETITET, A. et al. **HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers**. 2012. Available from Internet: <http://www.netlib.org/benchmark/hpl/>.

PICCOLI, G. et al. Compiler support for selective page migration in NUMA architectures. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES (PACT). **Proceedings...** [S.l.], 2014. p. 369–380.

PIELKE, R. A. et al. A Comprehensive Meteorological Modeling System- RAMS. **Meteorology and Atmospheric Physics**, v. 91, n. 1-4, p. 69–91, 1992.

PILLA, L. et al. **Improving Parallel System Performance with a NUMA-aware Load Balancer**. [S.l.], 2011.

RADOJKOVIĆ, P. et al. Thread Assignment of Multithreaded Network Applications in Multicore/Multithreaded Processors. **IEEE Transactions on Parallel and Distributed Systems (TPDS)**, v. 24, n. 12, p. 2513–2525, 2013.

RIBEIRO, C. P. et al. Improving memory affinity of geophysics applications on NUMA platforms using Minas. In: INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE COMPUTING FOR COMPUTATIONAL SCIENCE (VECPAR). **Proceedings...** [S.l.], 2010. p. 279–292.

RIBEIRO, C. P. et al. Memory Affinity for Hierarchical Shared Memory Multiprocessors. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING (SBAC-PAD). **Proceedings...** [S.l.], 2009. p. 59–66.

RIESEN, R. Communication Patterns. In: IEEE INTERNATIONAL PARALLEL & DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS). **Proceedings...** [S.l.], 2006.

RODRIGUES, E. R. et al. Multi-core aware process mapping and its impact on communication overhead of parallel applications. In: IEEE SYMPOSIUM ON COMPUTERS AND COMMUNICATIONS (ISCC). **Proceedings...** [S.l.], 2009. p. 811–817.

SHALF, J.; DOSANJH, S.; MORRISON, J. Exascale Computing Technology Challenges. In: HIGH PERFORMANCE COMPUTING FOR COMPUTATIONAL SCIENCE (VECPAR). **Proceedings...** [S.l.], 2010. p. 1–25.

SINGH, A. K. et al. Mapping on Multi/Many-core Systems: Survey of Current and Emerging Trends. In: DESIGN AUTOMATION CONFERENCE (DAC). **Proceedings...** [S.l.], 2013. p. 1–10.

SINGH, J. P.; ROTHBERG, E.; GUPTA, A. Modeling Communication in Parallel Algorithms: A Fruitful Interaction between Theory and Systems? In: ACM SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURES (SPAA). **Proceedings...** [S.l.], 1994. p. 189–199.

SISTARE, S.; VANDEVAART, R.; LOH, E. Optimization of MPI collectives on clusters of large-scale SMP's. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING (SC). **Proceedings...** [S.l.], 1999.

SOUTHERN, G.; RENAU, J. Deconstructing PARSEC Scalability. In: WORKSHOP ON DUPLICATING, DECONSTRUCTING AND DEBUNKING (WDDD). **Proceedings...** [S.l.], 2015.

THOZIYOOR, S. et al. A Comprehensive Memory Modeling Tool and Its Application to the Design and Analysis of Future Memory Hierarchies. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE (ISCA). **Proceedings...** [S.l.], 2008. p. 51–62.

TIKIR, M. M.; HOLLINGSWORTH, J. K. Hardware monitors for dynamic page migration. **Journal of Parallel and Distributed Computing (JPDC)**, v. 68, n. 9, p. 1186–1200, sep 2008.

TRÄFF, J. L. Implementing the MPI Process Topology Mechanism. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING (SC). **Proceedings...** [S.l.], 2002. p. 1–14.

TRAHAY, F. et al. EZTrace: a generic framework for performance analysis. In: INTERNATIONAL SYMPOSIUM ON CLUSTER, CLOUD AND GRID COMPUTING (CCGRID). **Proceedings...** [S.l.], 2011. p. 618–619.

UH, G.-R. et al. Analyzing dynamic binary instrumentation overhead. In: WORKSHOP ON BINARY INSTRUMENTATION AND APPLICATIONS (WBIA). **Proceedings...** [S.l.], 2006.

VALGRIND DEVELOPERS. **Cachegrind: a cache and branch-prediction profiler**. 2014. Available from Internet: <http://valgrind.org/docs/manual/cg-manual.html>.

VALGRIND DEVELOPERS. **Valgrind Documentation - Release 3.10.0**. [S.l.], 2014.

VAN DER WIJNGAART, R. F.; JIN, H. **NAS Parallel Benchmarks, Multi-Zone Versions**. [S.l.], 2003.

VERGHESE, B. et al. Operating system support for improving data locality on CC-NUMA compute servers. **ACM SIGPLAN Notices**, v. 31, n. 9, p. 279–289, 1996.

VERGHESE, B. et al. **OS Support for Improving Data Locality on CC-NUMA Compute Servers**. [S.l.], 1996.

VERSCHELDE, J. **MCS 572: Introduction to Supercomputing**. 2014. Available from Internet: <http://homepages.math.uic.edu/~jan/mcs572/index.html>.

VILLAVIEJA, C. et al. DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES (PACT). **Proceedings...** [S.l.], 2011. p. 340–349.

WANG, W. et al. Performance Analysis of Thread Mappings with a Holistic View of the Hardware Resources. In: IEEE INTERNATIONAL SYMPOSIUM ON PERFORMANCE ANALYSIS OF SYSTEMS & SOFTWARE (ISPASS). **Proceedings...** [S.l.], 2012.

WANG, Y.-T.; MORRIS, R. J. T. Load Sharing in Distributed Systems. **IEEE Transactions on Computers**, C-34, n. 3, p. 204–217, 1985.

WEISBERG, P.; WISEMAN, Y. Using 4KB page size for virtual memory is obsolete. In: 2009 IEEE INTERNATIONAL CONFERENCE ON INFORMATION REUSE AND INTEGRATION (IRI). **Proceedings...** [S.l.], 2009. p. 262–265.

WONG, C. S. et al. Towards achieving fairness in the Linux scheduler. **ACM SIGOPS Operating Systems Review**, v. 42, n. 5, p. 34–43, jul 2008.

WOODACRE, M. et al. **The SGI Altix 3000 Global Shared-Memory Architecture**. [S.l.], 2005.

WU, X. et al. Hybrid Cache Architecture with Disparate Memory Technologies. **ACM SIGARCH Computer Architecture News**, v. 37, n. 3, p. 34, 2009.

WULF, W. A.; MCKEE, S. A. Hitting the Memory Wall: Implications of the Obvious. **ACM SIGARCH Computer Architecture News**, v. 23, n. 1, p. 20–24, 1995.

ZHAI, J.; SHENG, T.; HE, J. Efficiently Acquiring Communication Traces for Large-Scale Parallel Applications. **IEEE Transactions on Parallel and Distributed Systems (TPDS)**, v. 22, n. 11, p. 1862–1870, 2011.

ZHOU, X.; CHEN, W.; ZHENG, W. Cache Sharing Management for Performance Fairness in Chip Multiprocessors. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES (PACT). **Proceedings...** [S.l.], 2009. p. 384–393.

ZIAKAS, D. et al. Intel QuickPath Interconnect - Architectural Features Supporting Scalable System Architectures. In: SYMPOSIUM ON HIGH PERFORMANCE INTERCONNECTS (HOTI). **Proceedings...** [S.l.], 2010. p. 1–6.