Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport

Joseph Antony, Pete P. Janes, and Alistair P. Rendell

Department of Computer Science
Australian National University
Canberra, Australia
alistair.rendell@anu.edu.au

Abstract. Modern shared memory multiprocessor systems commonly have non-uniform memory access (NUMA) with asymmetric memory bandwidth and latency characteristics. Operating systems now provide application programmer interfaces allowing the user to perform specific thread and memory placement. To date, however, there have been relatively few detailed assessments of the importance of memory/thread placement for complex applications.

This paper outlines a framework for performing memory and thread placement experiments on Solaris and Linux. Thread binding and location specific memory allocation and its verification is discussed and contrasted.

Using the framework, the performance characteristics of serial versions of `lmbench`, `Stream` and various BLAS libraries (ATLAS, GOTO, ACML on Opteron/Linux and Sunperf on Opteron, UltraSPARC/Solaris) are measured on two different hardware platforms (UltraSPARC/FirePlane and Opteron/HyperTransport). A simple model describing performance as a function of memory distribution is proposed and assessed for both the Opteron and UltraSPARC.

1 Introduction

Creation of scalable shared memory multiprocessor systems has been made possible by cc-NUMA (cache-coherent Non-Uniform Memory Access) hardware. This approach uses a basic building block comprising one or more processors with local memory and an interlinking cache coherent interconnect [5]. Unlike UMA (Uniform Memory Access) systems which comprise processors with identical cache and memory latency characteristics, NUMA systems exhibit asymmetric memory latency and possibly asymmetric bandwidths between its building blocks. On such platforms the operating system should consider physical processor and memory locations when allocating resources (i.e. memory allocation and CPU scheduling) to processes.

Y. Robert et al. (Eds.): HiPC 2006, LNCS 4297, pp. 338–352, 2006.
© Springer-Verlag Berlin Heidelberg 2006

Pthreads [6] and OpenMP [17] are two widely used programming models which target shared memory parallel computers. Both, however, were developed for UMA platforms, making no assumptions about the physical location of memory or where a thread is executing. Although there has been debate about the merit of adding NUMA extensions to these programming models, this issue is yet to be resolved [7]. In part the aim of the work presented here is to develop the tools and protocols required for performing memory and thread placement experiments that can be used to address this issue.

Linux and Solaris are two examples of operating systems that claim to be "NUMA aware". Exactly what this implies is not always well defined, but suffice it to say that both Solaris and Linux provide application program interfaces (API) that give the user some level of control over where threads are executed and memory is allocated [11,14] and perform some form of default NUMA aware placement of threads and data. While this is useful, for the programmer wishing to explore NUMA issues it is also useful to have functions that will identify the CPU currently being used by that thread, and the physical location that corresponds to an arbitrary (but valid) virtual address within an executing process.

In this paper we compare the support provided for thread and memory placement by Solaris and Linux, and also outline how a user can interrogate these runtime environments to determine actual thread and memory placements. Using this infrastructure the performance characteristics of two contemporary NUMA architectures – the UltraSPARC [20] using the FirePlane interconnect [4] and the Opteron [12] using HyperTransport [9] - are explored through a series of latency, bandwidth and basic linear algebra (BLAS) experiments.

A novel placement distribution model (PDM) which describes performance as a function of bandwidth and latency is presented and used to analyse performance results. The PDM uses directed graphs representing processor, memory and interconnect layout to aid in the enumeration of contention classes. The distribution of these contention classes permit qualitative analysis of performance data from NUMA platform experiments.

The paper is structured into the following sections – thread and memory placement on Solaris and Linux is discussed in section 2. The experimental hardware and software platforms used are described in section 3 while section 4 outlines the latency, bandwidth experiments and the placement distribution model. Section 5 covers related work and section 6 presents our conclusions.

2 Thread and Memory Placement

Conceptually, both Solaris and Linux are similar in their approach to abstracting underlying groupings of processors and memory based on latency. Yet, the mechanics of using the two NUMA APIs are quite different. Below we provide a brief review of Solaris thread and memory placement APIs, before contrasting this with the Linux NUMA support. We then consider placement verification for both Solaris and Linux.

2.1 Solaris NUMA Support

Solaris represents processor and memory resources as locality groups [11]. A locality group (`lgrp`) is a hierarchical DAG (Directed Acyclic Graph) representing processor-like and memory-like devices, which are separated from each other by some access-latency upper bound. A node in this graph contains at least one processor and its associated local memory. All the `lgrp`s in the system are enumerated with respect to the root node of the DAG, which is called the root `lgrp`. Two modes of memory placement are available, *next-touch*[1] and *random*[2]. The former is the default for thread private data, while the latter is useful for shared memory regions accessed by multiple threads as it can reduce contention.

A collection of APIs for user applications wanting to use `lgrp` information or provide memory management hints to the operating system is available through `liblgrp` [18]. Memory placement is achieved using `madvise()`, which provides advice to the kernel's virtual memory manager. The `meminfo()` call provides virtual to physical memory mapping information. We also note that memory management hints are acted upon by Solaris subject to resources and system load at runtime.

Threads have three levels of binding or affinity – *strong*, *weak* or *none* which are set or obtained using `lgrp_affinity_set()` or `lgrp_affinity_get()` respectively. Solaris' memory placement is determined firstly by the allocation policy and then with respect to threads accessing it. Thus there is no direct API for allocating memory to a specific `lgrp`, rather a first touch memory policy must be in place and then memory allocated by a thread that is bound to that specific `lgrp`. Within an `lgrp` it is possible to bind a specific thread to a specific processor by using the `processor_bind()` system call.

2.2 Linux NUMA Support

NUMA scheduling and memory management became part of the mainstream Linux kernel as of version 2.6. Linux assigns NUMA policies in its scheduling and memory management subsystems. Memory management policies include *strict*[3] allocation to a node, *round-robin*[4] memory allocations, and *non-strict preferred* binding to a node (meaning that allocation is to be *preferred* on the specified node, but should fall back to a default policy if this proves to be impossible). In contrast, Solaris specifies policies for shared and thread local data.

The default NUMA policy is to map pages on to the physical node which faulted them in, which in many cases maximises data locality. A number of system calls are also available to implement different NUMA policies. These system calls modify scheduling (`struct task_struct`) and virtual memory (`struct vm_area_struct`) related variables structures within the kernel.

---

[1] The next thread which touches a specific block of memory will possibly have access to it locally i.e. if remote memory is accessed it will possibly be migrated.

[2] Memory is placed randomly amongst the `lgrp`s.

[3] Memory allocation is to occur at a given node. It will fail if there is not enough memory on the node.

[4] Memory is dispersed equally amongst the nodes.

Relevant system calls include mbind(), which sets the NUMA policy for a specific memory area, set_mempolicy(), which sets the NUMA policy for a specific process, and the sched_setaffinity(), which sets a process' CPU affinity. Several arguments for these system calls are supplied in the form of bit masks, and macros, which makes them relatively difficult to use. For the application programmer a more attractive alternative is provided by the libnuma API. Alternatively, numactl is a command line utility that allows the user to control the NUMA policy and CPU placement of a entire executable[5].

Within libnuma useful functions include the numa_run_on_node() call to bind the calling process to a given node and numa_alloc_onnode() to allocate memory on a specific node. Similar calls are also available to allocate interleaved memory, or memory local to the caller's CPU. In contrast to the Solaris memory allocation procedure, numa_alloc modifies variables within the process' struct vm_area_struct and the physical location of the thread that performs the memory allocation is irrelevant. The libnuma API can also be used to obtain the current NUMA policy and CPU affinity. To identify NUMA related characteristics libnuma accesses entries in /proc and /sys/devices. This makes applications using libnuma more portable those that use the lower level system calls directly.

2.3 Placement Verification

Solaris provides a variety of tools[6] to monitor process and thread lgroup mappings – lgrpinfo, pmadvise, plgrp and pmap. The lgrpinfo tool displays the lgroup hierarchy for a given machine. The pmadvise tool can be used to apply memory advice to a running process. The plgrp tool can observe and affect a running thread's lgroup; it can also give a diagrammatic representation of the lgroup affinities. The pmap tool permits display of lgroups and physical memory mapping for all virtual address associated with a running process.

Although libnuma provides a means for controlling memory and process placement on Linux systems, it does not provide a means for determining where a given area of memory is physically located. A kernel patch that attempts to addresses this issue is provided by Per Ekman [15]. The patched kernel creates per-PID /proc entries that include, among other things, information about which node a process is running on, and a breakdown of the locations of each virtual memory region belonging to that process. While we found that this package was generally sufficient as a verification tool it involved having to check quickly the /proc entries while the program was running. We also found that under some circumstance the modified kernel failed to free memory after a process had terminated.

Based on the work of Ekman [15] we designed an alternative kernel patch that provides a system call and user level function to return the memory locations for each page in a given virtual memory range. This utility proved considerably more convenient as it could be called from within a running application.

[5] It can also be used to display NUMA related hardware configuration and configuration status.

[6] http://opensolaris.org/os/community/performance/numa/observability

3 Experimental Platforms

Two NUMA platforms were used in this work: a twelve processor Sun Ultra-SPARC V1280 [21] and a four processor AMD848 Opteron system based on the Celestica A8448 [3] motherboard. We now briefly outline the NUMA characteristics of each platform.

3.1 UltraSPARC/FirePlane

The V1280 has twelve 900MHz UltraSPARC III Cu processors each with a 32 Kb L1 instruction cache, 64 Kb L1 data cache and 8192 Kb L2 cache which is off-chip. The system contains three boards which hold four processors each and are linked using the FirePlane interconnect [4]. The system contains 8GB of memory per board giving a total of 24GB for the entire system. The three boards form a combined snooping based coherency domain. For larger systems, i.e. $>$ 24 processors, a directory based protocol is used at the point-to-point level.

A pair of processors and their associated memories are all linked using a Dual CPU Data Switch (DCDS), i.e. there are four separate data paths each running at 2.4 GB/s from processors or memories to the DCDS. The DCDSs can sustain 4.8 GB/s to the board data switch. Since memory on the boards is 16-way interleaved across a board, a peak of 6.4 GB/s per board is achieved. The point-to-point links among boards have a bi-directional bandwidth of 4.8 GB/s per board, approaching a peak of 9.6 GB/s for the whole system. Since the four processors on a board have similar memory access latencies, we will refer to it as one node. A schematic illustration of the V1280 is given in Figure 1 (a).

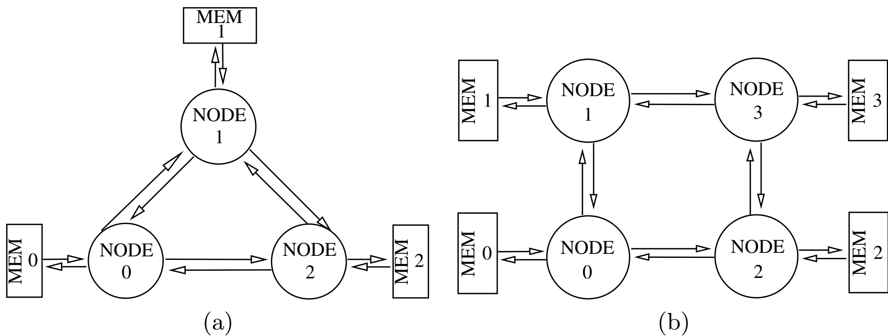

Fig. 1. (a) Schematic diagram of the V1280 UltraSPARC platform and (b) Celestica Opteron platform

3.2 Opteron/HyperTransport

The Opteron system contains four 2.2Ghz AMD848 processors each with a 64 Kb L1 data and instruction cache and a 1024 Kb L2 cache. The Celestica A8440 motherboard is configured with 2GB of memory per processor giving a total of 8GB for the entire system. The AMD848 Opterons have an on-chip memory

controller and uses coherent HyperTransport to link processor coherency traffic. The Opteron has two coherent HyperTransport links [9], each operating at 6.4 GB/s bi-directionally. The processors are arranged in a ring topology resulting in processors having at most two hops to reach the most distant processor. A schematic illustration of the Opteron system is given in Figure 1 (b).

3.3 Software Platform

While Solaris 10 was used on the V1280 system, the Opteron platform was capable of dual booting into either Solaris 10 or OpenSuSE 10. The Sun Studio 11 compilers were used on both Solaris platforms, while version 6.0 of the Portland Group compilers were used under Linux. Compiler flags for the highest optimisation levels were used on both compilers. To obtain accurate performance data the PAPI library [2] was used to access hardware performance counters under Linux whereas the `libcpc` [19] infrastructure was used under Solaris. The numeric libraries used under Linux are the ACML (version 3.0) from AMD, ATLAS (version 3.6) [23], GOTO BLAS [8] (version 1.00) while Sunperf (Sun Studio 11) was used under Solaris.

4 Results

This section discusses observed memory latency, serial memory bandwidth and parallel memory bandwidth for the two NUMA platforms.

4.1 Latency Characterisation

To determine the memory latency characteristics of the two platforms the `lm bench` [13] memory latency benchmark was modified to accept memory and thread placement parameters. Latencies to get data from level-one cache (L1) on the Opteron and UltraSPARC were measured as 3 and 2 cycles respectively, while accessing level-two cache (L2) took 20 cycles on both platforms. The latencies recorded for a thread bound to a particular node accessing memory at a specific location are given in Table 1. From these, the NUMA ratio[7] of the Opteron system is found to be 1.11 for one hop and 1.53 for two hops from any given processor, while on the V1280 there is only one NUMA level with a ratio of 1.2. While these are NUMA machines with low NUMA ratios, the emphasis of this paper is the sketching out and testing of the memory and thread placement framework with the view of extending it to NUMA systems with higher NUMA ratios.

4.2 Bandwidth Characterisation

To determine the memory bandwidth characteristics of the two platforms the `Stream` [10] benchmark was modified to accept memory and thread placement parameters. This benchmark performs four different vector operations, corresponding to vector copy, scale, add, and triad. On the Opteron system there are four

[7] NUMA ratio = $\frac{RemoteLatency}{LocalLatency}$

Table 1. Main Memory latencies (Cycles). `lmbench` uses a pointer chasing benchmark to determine memory latencies. Results were obtained for the Opteron and V1280 platforms by pinning a thread on a given node and placing memory on different nodes.

Thread Location	Memory Location: Opteron 0	1	2	3	V1280 0	1	2
0	225	250	250	345	220	265	265
1	250	225	345	250	265	220	265
2	250	345	225	250	265	265	220
3	345	250	250	225	–	–	–

nodes and four physically distinct memory banks, while on the UltraSPARC system there are three nodes and three memory banks. For a single thread it might be expected that the "best" possible `Stream` performance would be obtained when a thread is accessing vectors that are stored entirely in local memory. Conversely the "worst" possible performance would correspond to a thread accessing data stored in memory located as far away as possible.

Results for these two scenarios are given in Table 2. For the Opteron system running Solaris we find performance differences between best and worst memory placement that vary from a factor of 1.4 to 1.6. For Linux on the same platform we find a some what larger variation with factors between 1.09 to 2.35. On the V1280 system the effect is considerably less indicating relatively mild NUMA characteristics. (We note that the superior performance of the copy operation on the Opteron using Linux reflects the use of specialised instructions by the PGI compiler to perform the memory moves).

Table 2. Serial Stream bandwidths (GB/s) for the Opteron and V1280 systems. A single thread was pinned to a given node and had its memory placed on different nodes. Best and Worst refer to thread and memory placements which are expected to give the best and worst possible performance (See text for details).

	Opteron				V1280	
	Solaris		Linux		Solaris	
Test	Best	Worst	Best	Worst	Best	Worst
Copy	2.17	1.99	4.68	3.14	0.72	0.71
Scale	2.50	1.58	2.35	1.47	0.79	0.74
Add	2.75	1.17	2.55	1.54	0.83	0.81
Triad	2.24	1.51	2.44	1.52	0.85	0.79

The `Stream` benchmark was modified to create multiple threads, that concurrently ran separate instances of the original `Stream` benchmark. Results for this are presented in Table 3. In this case the worst case scenario on the Opteron would correspond to node 0's executing the `Stream` benchmark with all the data being serviced from memory 3, while the opposite happens on node 3, and nodes

1 and 2 are similarly exchanging data. Not surprisingly on both the Opteron and V1280 system the difference between good and bad memory placement has increased significantly over that observed for the serial benchmark.

Table 3. Parallel Stream bandwidths (GB/s). Threads were pinned to various nodes and had its memory placed locally ("Best") or remotely ("worst"). Four threads were run concurrently for the Opteron which twelve threads were run concurrently for the V1280 system.

	Opteron				V1280	
	Solaris		Linux		Solaris	
Test	Best	Worst	Best	Worst	Best	Worst
Copy	8.98	2.53	16.55	4.22	4.89	3.56
Scale	9.98	2.67	9.60	2.67	4.91	3.46
Add	10.85	2.85	10.33	2.94	5.22	3.57
Triad	9.17	2.68	9.87	2.96	5.14	3.71

4.3 Placement Distribution Model

In the above we considered "best" and "worst" case scenarios for the various `Stream` benchmarks. In the general case as well as on the Opteron system each vector or data quantity used in a `Stream` benchmark could be located in the memory associated with any one of the four available nodes. For the parallel add and triad benchmarks, on the Opteron system, this means that there are a total of $4^{12}*4!$ possible thread/memory combinations[8] while $4^8*4!$ copy and scale benchmarks are possible (add and triad benchmarks use 3 data quantities while copy and scale use 2 data quantities). Obviously evaluating the performance characteristics of each of these cases quickly becomes impossible for large NUMA systems. Thus, it is useful to develop a simple performance model which gives the *probability* of a given memory and thread placement experiments.

A placement distibution model (PDM) is developed to categorize the occurrence and type of possible placements. A directed graph of the NUMA platform is given to the model along with the data quantities used per thread. Figures 1 (a), (b) can be interpreted as graphs where links entering and exiting nodes are arcs. Traffic associated with each link can be modeled as weights along the links between nodes. Nodes are assumed to route traffic to their local memory controller or to other nodes along the most direct path. The model also assumes concurrent execution of all defined threads accessing its data quantities in tandem with other threads over the interconnect. This model can be used to characterise the communication requirements for any given memory placement experiment.

[8] A given data quantity could reside in 4 possible memory locations and each thread could run on 4 possible processors i.e. there are a total of 4^3 experiments for one thread and three data quantities. For all the 4 threads in the system there are $4^3 * 4 * 4^3 * 3 * 4^3 * 2 * 4^3 * 1 = 4^{12} * 4!$ possible combinations.

4.4 Placement Distribution Algorithm

An algorithm for the placement distribution model is presented in Algorithm 1. The PDM requires a graph $\mathbb{G}$, which represents the layout of memory $\mathbb{M}$, processor nodes $\mathbb{N}$ and a set $\mathbb{I}$ of ordered processor to memory or memory to processor data movements for a set of data quantities $\mathbb{D}$. These inputs are used to traverse over all possible configurations per thread of both thread and memory placement

Algorithm 1. The Placement Distribution Model

```
1: ℕ ← {node_1, node_2, ... , node_i}                 The set of all processor nodes
2: 𝕄 ← {mem_1, mem_2, ... , mem_j}                     The set of memory nodes
3: 𝕃 ← { link_1, link_2, ... , link_k}                The set of all links between nodes
4: 𝕋 ← { data_1, data_2, ... , data_l}                The set of data quantities
5: 𝔼 ← ℕ x 𝕄                                           Cartesian product denoting data movement
6: 𝔾 ← <𝔼,𝕃>                                           Graph 𝔾 representing memory and processor layout
7: 𝔻 ←{<x,y> | x ∈ 𝕋, y ∈ 𝕄}                          A data quantity x resides in memory location y
8: 𝕀 ← 𝔼 x 𝔻                                           Set of inputs for thread, memory placement
9: 𝕀 ≡ {<e, f> | e =<n,m>∈ 𝔼, f =<x,y>∈ 𝔻}
10: W(l) | l ∈ 𝕃                                       Weight matrix W
11: C(x,y)                                             Cost matrix C

Require: <n,m>∈ 𝔼
12: procedure OptPath(<n,m>)                           Optimal path from n to m where n,m ∈ 𝔼
13:                                                    Use appropriate alogrithm or heuristic
14:     return {<x,y> | x,y ∈ 𝕃}                       to get path between <n,m>
15: end procedure

Require: x ∈ 𝔻 ∀x ∈ ℚ
Require: <x,y>∈ 𝕃 ∀ <x,y>∈ ℙ
16: procedure FlowSize(ℚ, ℙ)                           Compute cost of moving data items across link ℙ
17:     cost ← 0
18:     for all (link ∈ ℙ) do
19:         for all (qty ∈ ℚ) do
20:             cost ← cost + | qty | ∗ W(link)
21:         end for
22:     end for
23:     return cost
24: end procedure

25: procedure ComputeDistribution
26:     ℚ' ← {x | x ∈ 𝔻}                               Set of data quantities of interest
27:     for all (i ∈ 𝕀) do                             Loop over input 𝕀 (i ≡<e,f>)
28:         links ← OptPath(e) where e ∈ i             Get the optimal path for a given e
29:         for all ((j ← links) ∧ (f ∈ i)) do         Loop over links and use f ∈<e,f>
30:             C(i,j) = C(i,j) + FlowSize(ℚ', j)
31:         end for
32:     end for
33: end procedure
```

for each data quantitity. A traversal implies data quantities are moved over a link and this entails a cost $W(l)$ per link l. Each traversal contributes to a cumulative cost entry in cost matrix C. Three procedures are defined in Algorithm 1 namely *OptPath*, *FlowSize* and *ComputeDistribution*. Procedure *OptPath* returns the optimal path, a set of ordered pairs of $<x,y>$ between two end points $<n,m>$ while procedure *FlowSize* computes a cost associated with moving data quantities contained in set $\mathbb{Q}$ over links contained in set $\mathbb{P}$ and procedure *ComputeDistribution* uses a set of data quantities as used per thread for all threads in set $\mathbb{Q}'$ and computes the cost for these data quantites for an ordered set of inputs $\mathbb{I}$.

A state machine was coded to perform walks along the links of graph $\mathbb{G}$, for all possible thread and memory placements given a specific processor/memory topology and data quantities. These walks model link traffic moving from a source node to a target node, traffic moving from a node to its local memory and traffic moving from one memory bank to another. In the event that there are two paths to the required destination of equal length, the traffic is split equally along each path. This assumption is made as a simplification to avoid complex specification of the underlying interconnect protocol. For example, if a placement dictates that Node 1 will be continuously accessing memory from Node 0, we increment variables belonging to each link along the route to record the quantity of the data movement. This results in a tuple holding values for link contention and node contention.

Using the PDM, for a given processor and memory layout, yields costs for thread and memory placement which are distributed in ranges which we term as *link contention classes*. The range of a link contention class gives the degree of contention at a node. For each contention class obtained from the PDM, 20 random configurations were generated i.e. thread and memory placement for all threads and data quantities which yields a link contention that lies in the range of all observed link contention classes. These placement configurations are subsequently used to perform copy and scale `Stream` measurements. In effect this process permits for a tractable analysis of possible performance characteristics for the benchmarks without resorting to running all experiments for all possible thread and memory placements.

Table 4 characterises the copy and scale `Stream` benchmarks according to the maximum level of contention on any given link. This table shows, for example, that on the Opteron system 51.9% of all possible memory placement configurations have link contentions greater or equal to 3 but less than 4, while 0.1% have a link contention of between 7 and 8. The ranges 3-4 and 7-8 are the *link contention classes*. The results show that on the Opteron system given random vector placement the probability of landing in a 3-4 link contention class is the highest, and within this class you might expect to see a performance degradation of about 20%. On the V1280 the effect is much less.

4.5 BLAS Experiments

Using the memory placement framework developed above, experiments were conducted for level 2 (DGEMV – matrix vector) and level 3 (DGEMM – matrix

Table 4. Copy and Scale (GB/s) Stream benchmark results for the placement distribution model. Contention classes denote the ranges of link contention for all the nodes in the system. %Freq gives the frequency of occurance of a given class in percent.

Contention		Solaris		Linux	
Class	%Freq	Copy	Scale	Copy	Scale
Opteron					
2-3	2.6	5.7	6.0	7.4	5.6
3-4	51.9	5.0	5.1	6.7	4.7
4-5	34.6	4.5	4.8	6.4	4.2
5-6	9.2	3.9	4.6	5.5	3.6
6-7	1.5	3.3	3.4	4.4	3.0
7-8	0.1	3.0	3.0	3.3	2.7
V1280					
08-12	10.3	4.0	4.0		
12-16	59.7	3.8	3.9		
16-20	24.8	3.7	3.8		
20-24	5.0	3.6	3.6		

multiply) BLAS operations. Results obtained for square matrices of dimension 1600 using `ACML` on the Opteron and `sunperf` on the V1280 are given in Table 5. In addition we also include results obtained from the triad `Stream` benchmark as these are representative of level 1 BLAS operations. The results show greatest NUMA effects on the Opteron system, where, as expected the variation is largest for triad, less for level 2 BLAS and almost unnoticeable for level 3 BLAS. This

Table 5. BLAS Stream Triad, Level 2 BLAS, Level 3 BLAS (GigaFlops) results for the placement distribution model. Results are averages for twenty random generated configurations per contention class. Tr = Triad.

Contention		Solaris			Linux		
Class	%Freq	Tr	L2	L3	Tr	L2	L3
Opteron							
3-4	1.9	0.5	1.6	15.3	0.5	1.5	15.6
4-5	38.1	0.4	1.4	15.2	0.4	1.4	15.6
5-6	38.2	0.4	1.5	15.2	0.4	1.4	15.6
6-7	16.0	0.4	1.4	15.2	0.4	1.3	15.6
7-8	5.0	0.3	1.3	15.2	0.3	1.3	15.6
8-12	3.4	0.3	1.1	14.9	0.3	0.8	15.6
V1280							
12-16	8.3	0.4	1.0	17.4			
16-20	48.3	0.3	1.0	15.8			
20-24	30.7	0.3	1.0	16.2			
24-28	10.2	0.3	1.0	17.4			
28-40	2.3	0.3	1.0	17.5			

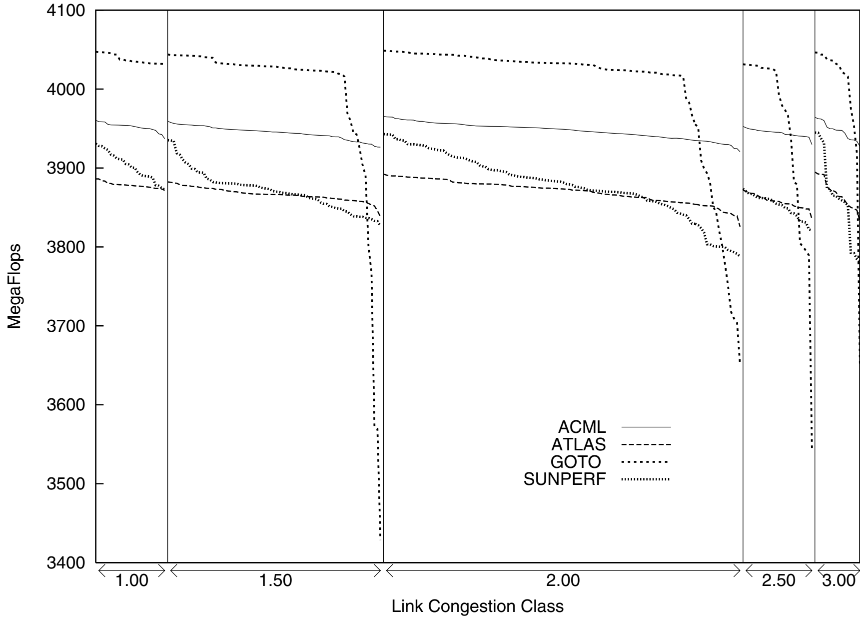

Fig. 2. Serial Opteron DGEMM ($C = A * B$) performance (MegaFlops) for square matrices of dimension 1600 using ACML, ATLAS, GOTO and Sunperf libraries. A total of 256 experiments were run for all possible thread and memory placements for one thread and three data quantities A, B and C.

reflects the fact that a well written DGEMM will spend most of its time working on data that is resident in the level 2 cache, but this is not possible for level 1 or level 2 BLAS where data must be streamed from memory to processor.

In Figure 2 we present floating point performance for serial DGEMM on the Opteron system using four different BLAS libraries: i) ACML, ii) ATLAS , iii) GOTO and iv) Sunperf. With three matrices and four different nodes on the Opteron system there are a total of 256 different thread and memory placement permutations. These have been ordered according to maximum contention on a given link and then sorted by performance observed within that group. As this is a serial matrix multiply, the link contention ranges from 1 to 3. Memory contention of 1 occurs when the three matrices are located on adjacent nodes with the compute thread is bound to the central node, i.e. the contention on the network is actually reduced compared to the case when all three matrices are local to the node accessing them (contention value of 3). Interestingly in some cases the best performance is obtained for a link contention of 2 indicating that on an idle machine non-local placement of some data quantities may be advantageous if it leads to enhanced overall memory bandwidth. The performance also shows considerable fine structure, especially for GOTO BLAS [8] which for most of the time exhibits the best performance, but in some cases also shows the worst

performance. At this point we believe the reason for these sudden performance drops (of ≈16%) is cache line conflicts arising from slightly different memory placements within a node.

5 Related Work

Brecht [1] evaluates the importance of placement decisions on NUMA machines with different NUMA ratios. Application placement which mirrored hardware is beneficial for application performance and its importance increased with the NUMA ratio.

Robertson and Rendell [16] quantify the effects of memory bandwidth and latency on the SGI Origin 3000 using `lmbench` and `stream`. Using a 2D heat diffusion application, they stress the importance of good thread and memory placement and show that relying on the operating system for thread and memory placement is not always optimal.

Tikir and Hollingsworth [22] use link counters and a bus analyzer, on the SunFire 6800 system to effect transparent page migration, without modification to the operating system or application code. They are able to improve execution time of benchmarked applications by 16%. This is achieved by using a combination of hardware counters, runtime instrumentation and `madvise()`.

6 Conclusions

The support for thread binding and memory placement provided by Solaris and Linux has been outlined and contrasted. For Linux, the kernel was modified in order to provide a user API that could be used to verify binding and determine physical memory placement from a user supplied virtual address. Using the various thread and memory placement APIs, a framework was outlined for performing NUMA performance experiments. Detailed measurements of the latency, bandwidth and BLAS performance characteristics of two different hardware platforms were undertaken. These showed the Opteron system to be "more NUMA" than the Sun system, despite the fact that it had only 4 processors. To assist in the analysis of the performance data, a simple placement distribution model of the NUMA characteristics for the two platforms was outlined. The PDM uses directed graphs to represent processor, memory and interconnect layout. It was found that if multiple level 1 or level 2 BLAS operations are run in parallel on the Opteron system performance differences of up to a factor of two were observed depending on memory and thread placement. For level 3 BLAS, differences are much smaller as there is much better re-use of data from level 2 cache.

The use of the PDM shows node local allocation of memory is not always the best strategy for the DGEMM kernel. The best peak results were obtained for a link contention of 2 i.e. non-local placement of data. This highlights the benefits of user-level discovery, at runtime, of processor and memory topologies

and the use of this knowledge within the application to effect thread and memory placement specific to its needs.

Acknowledgments

This work was possible due to funding from the Australian Research Council, Gaussian Inc. and Sun Microsystems Inc. under ARC Linkage Grant LP0347178. We also wish to thank Peter Strazdins for his helpful suggestions and comments; Alexander Technology for access to the Opteron system.

References

1. T. Brecht. On the Importance of Parallel Application Placement in NUMA Multiprocessors. In *Proceedings of the Fourth Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 1–18, 1993.
2. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.
3. Celestica Inc. AMD A8440 4U 4 Processor SCSI System. `http://www.celestica.com/products/A8440.asp`.
4. A. Charlesworth. The Sun Fireplane System Interconnect. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*. ACM Press, New York, New York, USA, November 2001.
5. D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, California, USA, 1999.
6. David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Professional, 1997.
7. Dimitrios Nikolopoulos and Theodore Papatheodorou et. al. Leveraging Transparent Data Distribution in OpenMP via User-Level Dynamic Page Migration. In M. Valero, K. Joe, M. Kitsuregawa, and H. Tanaka, editors, *ISHPC*, volume 1940 of *Lecture Notes in Computer Science*, pages 415–427. Springer, 2000.
8. K. Goto and R. A. van de Geijn. Anatomy of High-Performance Matrix Multiplication. *In submission to ACM Transactions on Mathematical Software*, 2006.
9. Jay Trodden and Don Anderson. *HyperTransport System Architecture*. Addison-Wesley Professional, 2003.
10. John McCalpin. Stream: Sustainable memory bandwidth in high performance computers. `http://www.cs.virginia.edu/stream`.
11. Jonathan Chew. Memory Placement Optimisation. `http://www.opensolaris.org/os/community/performance/mpo_overview.pdf`.
12. C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway. The AMD Opteron Processor for Multiprocessor Servers. *IEEE Micro*, 23(2):66–76, 2003.
13. L. W. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.
14. Novell. A NUMA API for Linux. `http://www.novell.com/collateral/4621437/4621437.pdf`.
15. Per Ekman. Linux kernel memory-to-node mappings. `http://www.pdc.kth.se/~pek/linux/NUMA/`.

16. N. Robertson and A. P. Rendell. OpenMP and NUMA Architectures I: Investigating Memory Placement on the SGI Origin 3000. In P. M. A. Sloot, editor, *International Conference on Computational Science*, volume 2660 of *Lecture Notes in Computer Science*, pages 648–656. Springer, 2003.
17. Rohit Chandra and Ramesh Menon et. al. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.
18. Sun Microsystems. Solaris 10 : Extended Library Functions. `http://docs.sun.com/app/docs/doc/817-0679`.
19. Sun Microsystems. Solaris 10 : Programming Interfaces Guide. `http://docs.sun.com/app/docs/doc/817-4415`.
20. Sun Microsystems. *UltraSPARC III Cu User's Manual*. Sun Microsystems, Santa Clara, California, USA, January 2004. Version 2.2.1.
21. Sun Microsystems Inc. The Sun Fire V1280 Server Architecture. `http://www.sun.com/servers/midrange`, November 2002.
22. M. M. Tikir and J. K. Hollingsworth. Using Hardware Counters to Automatically Improve Memory Performance. In *SC*, page 46. IEEE Computer Society, 2004.
23. R. C. Whaley, A. Petitet, and J. Dongarra. ATLAS. *Parallel Computing*, 27 (1-2):3–35, 2001.