Scalable Computing: Practice and Experience
Volume 10, Number 4, pp. 429–441. http://www.scpe.org

ISSN 1895-1767
© 2009 SCPE

PERFORMANCE STUDY OF THE FIRST THREE INTEL MULTICORE PROCESSORS

AMI MAROWKA*

Abstract. The transition from sequential computing to parallel computing represents the next turning point in the way software engineers design and write software. This paradigm shift leads the integration of parallel programming standards for high-end shared-memory machine architectures into desktop programming environments. In this paper we present a performance study of these new systems. We evaluate the performance of an OpenMP shared-memory programming model that is integrated into Microsoft Visual Studio C++ 2005 and Intel C++ compilers on a multicore processor. We benchmarked using the NAS OpenMP high-level applications benchmarks and the EPCC OpenMP low-level benchmarks. We report the basic timings and runtime profiles of each benchmark and analyze the running results.

Key words: multicore, openMP, NPB, micro-benchmarks

1. Introduction. For many years parallel computers have been used only by an exclusive scientific niche. Only universities and research institutions backed by government budgets or funded by multi-billion-dollar companies could afford to purchase state-of-the-art parallel machines. Multiprocessor machines are very expensive and demand expertise in system administration and programming skills. Parallel computing therefore remains a specialized field of an exclusive community.

Now, two complementary technologies bring parallel computing to the desktop. On the hardware side is the multicore processor for desktop computers, and on the software side is the integration of the OpenMP parallel programming model into Microsoft Visual C++ 2005. These technologies promise massive exposure to parallel computing that nobody can ignore, thus making a technology shift unavoidable.

The dual-core processors first appeared on the market six years ago [1]. The chip makers Sun and IBM were the first: Sun introduced the Microprocessor Architecture for Java Computing and IBM launched the Power 4 dual-core processor. These processors, like their predecessors, were expensive and optimized for special-purpose computing with intensive tasks running on high-end servers. The greatest change came with the dual-core processors that AMD and Intel launched in 2005. Designed for desktop computers, these processors caused prices to drop; thus, a desktop computer with a dual-core processor can now be bought for less than $500. This price is affordable for students and computer science departments alike. But dual-core processors are only the beginning. The chip makers are now working on the next generation of multicore processors that will contain 4, 8, and 16 cores on a single die. Unfortunately, writing a parallel code is more complex than writing a serial code [2]. Parallel programming is extremely difficult. This is where the OpenMP programming model comes into the picture [3]. OpenMP helps developers to create multithreaded applications more easily while retaining the look and feel of serial programming.

The extra development effort and code complexity of parallel programming give rise to an obvious question– Is it worthwhile? The best way to answer this question is by benchmarking. This paper presents a performance study of OpenMP shared-memory programming model [3] that was integrated into Microsoft Visual Studio C++ 2005 and Intel C++ compilers on multicore processors. The benchmarking was conducted using the NAS OpenMP parallel benchmark suite [4] with different sizes of input classes, and the EPCC OpenMP directives benchmarks [7, 12]. We report the basic timings and runtime profiles of each benchmark and analyze the running results. A preliminary conference version paper of the study presented in this paper is [14]. This paper presents a completed study and includes details and materials that have been studied since the first version, such as related-works section; more tables and figures; more detailed tables and figures; benchmarking of Intel Quad-core machine; and clearer presentation of comparison between different architectures by clock cycles.

The rest of this paper is organized as follows. Section 2 presents related work. In Sections 3, 4, and 5 we provide brief overviews of the OpenMP, NPB benchmarks, and EPCC micro-benchmarks respectively. Section 6 is an in-depth analysis of the benchmarks results and Section 7 presents our conclusions.

2. Related Work. Multicore processors are ubiquitous and therefore they are studied intensively from many aspects. Many chipmakers offer today multicore processors with different architectures. The work presented in this paper covers the first generations of multicore processors from Intel. The performances of these

*Department of Computer Science, Shenkar College of Engineering and Design, Israel amimar2@yahoo.com processors are compared by using OpenMP NPB benchmarks and the EPCC micro-benchmarks on MS Windows operating system.

Chunhua Liao et al [15] reported on experiments of OpenMP on Sun Fire V490 with Chip Multiprocessing (Solaris 10 operating system) and a Dell Precision 450 workstation with Simultaneous MultiThreading(SMT) technology (Linux kernel 2.6.3 SMP(Symmetric Multi-Processor) operating system). For this study they used EPCC Microbenchmark suite, subsets of the benchmarks in SPEC OMPM2001 and the NAS parallel benchmark 3.0 suites. The main conclusions of this study are that a straightforward OpenMP implementation for traditional SMP architecture may not achieve good scalability on the Xeon system because of memory bandwidth bottleneck and competition for the shared computing resources. They found that a HyperThreading-aware OS is important for maintaining load balance and efficiently utilizing the resources of an SMT system and that the EPCC micro-benchmarks results also indicated that the overhead of OpenMP synchronization implementation in a SMT system is higher than that in an SMP system.

Kent Milfeld et al [16] studied the performance of two dual-core processors, Intel dual-core Woodcrest (RedHat OS) and AMD dual-core Opteron (SuSE OS). In this work the costs of creating Pthreads and OpenMP thread were investigated and the performance of thread/process scheduling and affinity in multi-cache systems and the thread synchronization methods and costs were evaluated. Also, the L2 Cache Characteristics (latency) and the impact of a large-page memory on the performance of four NPB-MPI benchmarks were studied. The main conclusions of the authors are that applications need to be optimized for CLP, but it is still uncertain which cache system, independent or shared, will be best. Moreover, shared L2 cache suffers from contention and low level synchronization methods for multiple cores and thus will need to be implemented in thread control at the user-program level to assure locality of the data. Nevertheless, independent cache systems are contention free at the L2 cache level, and are probably more ideal for MPI codes, which employ non-shared paradigms.

Furlinger Karl et al [17] studied the scalability characteristics of medium and large variants of the SPEC OpenMP benchmarks on large-scale shared memory multiprocessor machines using their own OpenMP profiling tool, ompP. The experiments were conducted on 2 to 32 processors on a 32 processor SGI Alitx 3700 with the medium variant while the large variant were tested on 32 to 128 processors, with increments of 16, of a larger Altix 4700 machine. Four overheads categories used to evaluate the scalability: (S) corresponds to synchronization overhead, (I) represents overhead due to imbalance, (L) denotes limited parallelism overhead and (M) signals thread management overhead. The results show that 4 of 7 medium applications and 3 of 5 large applications achieved poor scalability due to load imbalance, thread management overhead, and small parallel loops.

Grant and Afsahi [18] studied the optimal operating configuration of Hybrid chip multithreaded SMPs for scientific applications and to identify the shared resources that might become a bottleneck to performance under the different hardware configurations. The authors investigated a two-way dual-core Hyper-Threaded (HT) Intel Xeon SMP server under single program and multi-program multithreaded workloads using the NAS OpenMP benchmark suite. The experiments were conducted on a Dell PowerEdge 2850 SMP server with Red Hat Linux Enterprise WS 4.1 distribution with Kernel 2.6.9-11 while LMbench tool used for measuring the L1, L2, and main memory latencies of the processor. The performance results indicate that in the single-program case, the CMP-based SMP and CMT-based SMP configurations have the highest average speedup across all of the applications. The most efficient architecture is a single HT-enabled dual-core processor that is almost comparable to the performance of a 2-way dual-core HT disabled system.

Curtis-Maury et al [19] evaluated the performance of OpenMP applications on simulated CMP and SMT architectures using Simics simulation platform. The simulations evaluated the performance of NAS Parallel Benchmarks suite. The authors found that the high level of resource sharing in SMTs results in performance complications, should more than 1 thread be assigned on a single physical processor. CMPs, on the other hand, are an attractive alternative. Their results show that the exploitation of the multiple processor cores on each chip results in significant performance benefits. Moreover, the execution of multiple threads on each processor is more efficient and predictable on CMPs than it is on SMTs due to the higher degree of resource isolation, which results in fewer conflicts between threads co-executing on the same processor. Although adaptive run-time techniques can improve the performance of OpenMP applications on SMTs, inherent architectural bottlenecks hinder the efficient exploitation of these processors.

3. OpenMP Programming Model.

OpenMP is a tool for writing multi-threaded applications in a shared memory environment [3]. It consists of a set of compiler directives and library routines. The compiler generates a multi-threaded code based on the specified directives. OpenMP is essentially a comparatively recent standardization SMP (Symmetric Multi-Processor) development and practice. By using OpenMP, it is relatively easy to create parallel applications in FORTRAN, C, and C++. Compiler and third party applications support is becoming more common.

An OpenMP program begins with a single thread of execution called the master thread. The master thread spawns teams of threads in response to OpenMP directives, which perform work in parallel. Parallelism is thus added incrementally: the serial program evolves into a parallel one. OpenMP directives are inserted at key locations in the source code. These directives take the form of comments in FORTRAN and pragmas in C and C++. The compiler interprets the directives and creates the necessary code to parallelize the indicated tasks/regions. The parallel region is the basic construct that creates a team of threads and initiates parallel execution. Most OpenMP directives apply to structured blocks, which are blocks of code with one entry point at the top and one exit point at the bottom. The number of threads created when entering parallel regions is controlled by the value of the environment variable OMP_NUM_THREADS. The number of threads can also be set by a function call from within the program, which takes precedence over the environment variable. It is possible to vary the number of threads created in subsequent parallel regions. Each thread executes the block of code enclosed by the parallel region.

In general, there is no synchronization between threads. Different threads may reach the end of the parallel region at different times. OpenMP does provide constructs for synchronization, but the code should not be written in such a way that its output depends upon different threads executing statements at particular times. OpenMP provides a number of constructs for thread synchronization and coordination, among them critical, atomic, barrier, and master. These are sufficient for many needs, but OpenMP also provides a set of runtime thread-locking functions that can be used for fine control. When all threads reach the end of the parallel region, all but the master thread go out of existence and the master continues alone. The OpenMP directive clauses—Private, Shared, and Default—control whether the listed variables are shared among different threads or are private (local) to each thread.

OpenMP provides several constructs for sharing work among threads in a team. These are: Parallel for/do, Parallel Sections, Workshare, and Single directive. These constructs are placed inside an existing parallel region. The result is to distribute execution of associated statements among the existing threads. A number of environmental variables may be set to control aspects of OpenMP execution; for example, the number of threads, loop scheduling, and the enabling of nested parallelism and dynamic adjustment of the number of threads.

OpenMP also provides a number of routines that may be called from within one's code. These may be used to get and set the number of threads, enable or disable dynamic thread allocation, check whether the code is executing in parallel, etc. Changes in the runtime environment made by these routines take precedence over the corresponding environment variables.

4. NAS Parallel Benchmark. We used the NPB OpenMP-C benchmark suite, based on NPB 2.3-serial version, to evaluate the OpenMP performance on our multicore machines. Since the official OpenMP version of the NBP benchmarks from NASA is written only in FORTRAN we used the NPB OpenMP-C version that was developed as part of the Omni project [6]. The NAS Parallel Benchmarks (NPB) [4, 5] was devised by the Numerical Aerodynamic Simulation Program of NASA for the performance analysis of highly parallel computers. The NPB are valuable since they are rigorous and close to real-life needed applications. The NPB consist of five kernels and three simulated applications. They all compute or simulate different algorithmic and computational aspects of aerodynamic applications. For most of the kernels it is possible to select the problem size. Sometimes the problem sizes are called: Class S or T (12x12x12), Class W (24x24x24), and Class A 64x64x64).

The following is a brief description of the five kernels we used in our work.

Kernel EP. In the embarrassing parallel benchmark, two-dimensional statistics are accumulated from a large number of Gaussian pseudo-random numbers, which are generated according to a particular scheme that is well suited for parallel computation.

Kernel MG. The MG (Multi-grid) benchmark is a simplified multi-grid kernel, which solves a 3-D Poisson PDE. The Class W problem uses the same size grid as Class S but has a greater number of inner loop iterations.

Kernel CG. In the CG (Conjugate Gradient) benchmark, a conjugate gradient method is used to compute an approximation to the smallest eigen value of a large, sparse, symmetric positive definite matrix. This kernel is typical of unstructured grid computations applications.

Table 5.1
The Tested Multicore Machines

Platform	No. of Cores	Clock(GHz)	L1 Cache	L2 Cache	Memory
Intel Pentium D 820	2	2.8	2x16KB	2x1MB	512MB
Intel Core 2 Duo E6300	2	1.86	2x32KB	1x2MB	1GB
Intel Core 2 Quad Q6600	4	2.4	4x32KB	2x2MB	512MB

Kernel FT. In the FT (3-D FFT PDE) benchmark, a 3-D partial differential equation is solved using FFTs. This kernel performs the essence of many spectral methods. This benchmark is somewhat unique in that computational library routines may be legally employed.

Kernel IS. The IS (Integer Sort) benchmark tests a sorting operation that is important in particle method codes. This type of application is similar to particle-in-cell applications of physics, wherein particles are assigned to cells and may drift out. The sorting operation is used to reassign particles to the appropriate cells.

The three simulated applications we used are as follows.

BT is a simulated CFD application that uses an implicit algorithm to solve 3-dimensional (3-D) compressible Navier-Stokes equations. The finite differences solution to the problem is based on an Alternating Direction Implicit (ADI) approximate factorization that decouples the *x, y,* and *z* dimensions. The resulting systems are Block-Tridiagonal of 5x5 blocks and are solved sequentially along each dimension.

SP is a simulated CFD application that has a similar structure to BT. The finite differences solution to the problem is based on a Beam-Warming approximate factorization that decouples the *x, y,* and *z* dimensions. The resulting system has Scalar Pentadiagonal bands of linear equations that are solved sequentially along each dimension.

LU is a simulated CFD application that uses the symmetric successive over-relaxation (SSOR) method to solve a seven-block-diagonal system resulting from finite-difference discretization of the Navier-Stokes equations in 3-D by splitting it into block Lower and Upper triangular systems.

5. EPCC Microbenchmarks. The EPCC micro-benchmark suite is a set of benchmarks that measure the overhead incurred by OpenMP compiler directives of a specific OpenMP implementation [7, 12]. Three classes of overhead can be measured by the EPCC micro-benchmark suite: synchronization, loop scheduling, and array operations. The current release supports the OpenMP 2.0 standard. The overhead cost incurred by a specific compiler directive is measured by comparing the sequential execution time of a section code containing the compiler directive, and the parallel execution time of the same code. The measurements are repeated a few times for statistical stability. By using the EPCC micro-benchmark, the developer is able to compare the relative efficiency of different implementations of OpenMP running on the same platform; choose the more efficient construct of two semantically equivalent; and predict the overall performance of an application. Although there are other tools in the market that were developed for similar purposes as EPCC, such as ompP [10] that performs overhead analysis, and Sphinx [13], EPCC software is considered the de-facto standard of its kind.

6. Experimental Results. We tested the performance of the EPCC micro-benchmarks and the NAS OpenMP kernels and applications on multicore machines. The list of the tested platforms is shown in Table 1. On the software side we used the OpenMP version 2.0 of Intel C++ OpenMP compiler 11.0 under the XP operating system. All the benchmarks were also compiled by Microsoft Visual Studio C++ 2005 and evaluated on the above multicore machines. However, the differences between the results obtained by using the Microsoft and Intel compilers are negligible, so they are not shown here. Only the results obtained by Intel compiler are shown here. All the measurements shown are in units of Kilo-Clock-Cycles (KCC) and Giga-Clock-Cycles (GCC) for better comparison between different machine architectures. KCC is thousands of clock-cycles per second and GCC is billions of clock-cycles per second.

First, we measured the overhead cost of the OpenMP directives by using the EPCC micro-benchmarks. In general, it is an important practice to start parallel application benchmarks by examining the overhead incurred by the primitive parallelism functions used by the parallel programming model on the testbed machine. This way the programmer has a priori knowledge of the effects of various possible overhead sources on the total performance of the applications. Tables 2-6 and figures 1-5 show results measured on the Intel Pentium D, Intel

Table 6.1
Synchronization overheads(in KCC) of 1, 2 and 4 threads on Pentium D, Core 2 Duo and Core 2 Quad machines

	Pentium D			Core 2 Duo			Core 2 Quad		
Num of Threads	1	2	4	1	2	4	1	2	4
Parallel	2.8	37.3	234	1.3	16.1	65	1.2	46.6	45.2
for	0.2	14.5	292	0.1	6.2	65	0.1	11.1	12.2
Parallel-for	2.9	39.4	221	1.3	15.1	64	1.3	48	45.5
Barrier	0.2	14.0	288	0.01	6.2	63	0.02	10.9	11.7
Single	0.1	9.6	266	0.09	5.3	2.0	0.07	11.9	11.4
Critical	0.2	1.0	2.0	0.13	0.93	0.5	1.2	1.0	14.4
Lock-Unlock	0.2	1.4	2.5	0.16	0.93	0.8	1.3	1.8	10.6
Ordered	0.5	27.1	27.1	0.24	10.7	15.0	0.25	23.5	20.3
Atomic	0.1	0.3	0.4	0.07	0.22	0.20	0.07	0.3	1.0
Reduction	3.0	40.2	260	0.13	16.1	17.0	1.2	50	52

Core 2 Duo and Intel Core 2 Quad machines while running EPCC micro-benchmarks compiled by Intel C++ compiler 9.1.

Table 2 shows the OpenMP synchronization overheads measured by running the EPCC micro-benchmarks with 1, 2 and 4 threads. Figure 1 is a bar chart of the OpenMP synchronization overheads measured by running the EPCC micro-benchmarks with two threads. First, it can be observed from figure 1 that the overhead cost is less than 50K cycles in all the cases. This low overhead has a negligible effect on the NAS applications performance that will be discussed later. Second, the OpenMP directive overheads on the Intel Core 2 Duo are up to 50% less than the overheads incurred by the directives on the Intel Pentium D and the Intel Core 2 Quad. This improvement is mainly due to the shared L2 cache memory architecture of the Intel Core 2 Duo processor compared to the distributed L2 cache memory of the Intel Pentium D. The Intel Core 2 Quad processor contains two separate pairs of dual-core with shared L2 cache each. However, when two threads are spawned, the operating system maps each one of them to a different pair of dual-core. The physical separation of four cores to two pair of dual-core is the cause of the increase in the synchronization costs as shown in figure 1 and Table 2. On the other hand, when four threads are invoked on a quad-core processor, the operating system maps each one of the four threads to a different core and thus there is no competition between the threads on the available cores. The result is less overhead due to better scheduling as can be shown in Table 2. For example, the barrier synchronization overheads on the Intel Pentium D and the Intel Core 2 Duo (which are dual-core processors) with four threads are 288 KCC and 63 KCC respectively, while the barrier synchronization overhead on the Intel Core 2 Quad with four threads is only 11.7 KCC. We will elaborate on these architectures later. Third, the overhead of a single core is relatively high. Thus, it is better for single threaded applications to be compiled with the OpenMP option set to off.

Further analysis of the results leads to the following conclusions: the combined directive Parallel-For is more efficiently used than the Parallel and the For directives, which are used separately; it costs less to use the Critical directive than to use the Lock-Unlock pair directives; the Barrier and the Single directives have a relatively low overhead; the Order and the Reduction clauses have relatively high costs, as can be expected; and, finally, the overhead of the Atomic directive is negligible and thus is recommended for use, where it is possible, instead of the Critical or the Lock-Unlock directives. We omit the discussion on scalability with respect to the number of cores because it is useless to do such an analysis when the machines have only two and four cores.

Table 3 shows the Array (or privatization) overheads of four clauses: Private, Firstprivate, CopyPrivate, and Copyin for 1, 2 and 4 threads on the tested machines and figure 2 is a bar chart of the results for the case of two threads. First, it can be observed again that the Intel Core 2 Duo processor presents lower overheads than the Pentium D and the Core 2 Quad processors when using two threads. However, when using four threads the Intel Core 2 Quad processor is more efficient. For example, to access a private array on the Intel Pentium D and the Intel Core 2 Duo with four threads costs 160 KCC and 98 KCC respectively, while it costs only 36.2 KCC on the Intel Core 2 Quad with four threads. Moreover, the results show that the Private, Firstprivate, and Copyin clauses are incurred acceptable overhead costs for the array allocation process. The CopyPrivate demonstrates excellent performance and a negligible overhead cost that enables super-efficient inter-threads communication.

Table 6.2
Array privatization overheads(in KCC) of 1, 2 and 4 threads on Pentium D, Core 2 Duo and Core 2 Quad machines.

	Pentium D			Core 2 Duo			Core 2 Quad		
Num of Threads	1	2	4	1	2	4	1	2	4
Private	2.85	37.9	160	2.0	22.8	98	1.4	52	36.2
FirstPrivate	2.8	44.4	274	2.0	22.4	100	1.4	53.9	35.7
CopyPrivate	0.2	0.28	0.28	0.05	0.08	0.05	0.05	0.08	0.05
CopyIn	2.8	51.1	211	2.0	24.0	102	1.5	52.9	100

Table 6.3
OpenMP Loop Scheduling overheads (in KCC) of 1, 2 and 4 threads on Intel Pentium D machine.

	Threads	Pentium D							
Chunk size		1	2	4	8	16	32	64	128
Static	1	0.8	0.2	0.05	0.2	0.03	0.06	0.05	0.05
	2	16.8	15.3	18.6	20.5	15.6	21	15.2	15
	4	294	290	296	294	292	292	292	292
Dynamic	1	25	12	8.6	6.3	3.7	3.1	2.6	1.8
	2	92	50	40	30	25	23	20	19
	4	425	352	327	308	300	294	288	282
Guided	1	0.22	2.5	2.4	2.2	2.3	1.8	2.3	2.4
	2	27	25	24	23	22	23	19	18
	4	305	302	300	300	297	296	295	294

Table 6.4
OpenMP Loop Scheduling overheads (in KCC) of 1, 2 and 4 threads on Intel Core 2 Duo machine.

	Threads	Core 2 Duo							
Chunk size		1	2	4	8	16	32	64	128
Static	1	0.07	1.54	1.8	1.9	2.0	2.5	3.0	4.0
	2	7.4	6.2	6.2	6.4	7.5	7.5	6.0	6.2
	4	74	74	74	74	74	74	74	74
Dynamic	1	16.0	8.0	4.0	2.0	1.2	2.7	0.65	0.55
	2	39	21	15	12	10	8.7	8.2	8.7
	4	138	104	87	81	79	74	72	72
Guided	1	0.5	0.5	0.5	1.3	0.5	0.5	0.5	0.94
	2	10	8.7	9.6	9.3	8.2	7.5	7.5	7.5
	4	80	80	80	79	80	80	80	80

Table 6.5
OpenMP Loop Scheduling overheads (in KCC) of 1, 2 and 4 threads on Intel Core 2 Quad machine.

	Threads	Core 2 Quad							
Chunk size		1	2	4	8	16	32	64	128
Static	1	0.2	0.3	0.4	0.5	0.5	0.6	0.6	2.4
	2	10.3	10.0	9.8	7.8	7.0	10.1	9.9	9.8
	4	12.8	11.6	11.7	11.0	10.8	11.0	11.0	26.4
Dynamic	1	15.7	9.5	6.3	4.7	1.4	3.4	3.2	0.6
	2	89	25	13.3	17.4	15.8	10.5	11.4	13.6
	4	391	182	88	28.4	20.2	18.5	17.1	16.6
Guided	1	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
	2	16.4	10.5	12.5	14.3	15.7	8.8	13.5	12.0
	4	24.6	23.0	21.5	19.3	18.4	16.6	15.8	15.2

TABLE 6.6

Performance (in GCC) of EP, FT, MG, CG, BT, SP, and LU benchmarks for 1, 2 and 4 threads, and problem class W on Pentium D, Core 2 Duo and Core 2 Quad machines.

	Pentium D						
Threads	EP	FT	MG	CG	BT	SP	LU
1	33.68	2.57	2.54	2.35	33.76	99.96	66.05
2	16.66	2.52	1.7	2.18	29.7	146.46	37.12
4	46.64	2.04	2.85	3.19	37.91	1304	3967
	Core 2 Duo						
1	15.14	1.24	1.48	1.45	14.32	47.69	35.4
2	7.75	1.19	0.98	1.39	13.24	64.43	20.68
4	7.7	0.98	1.17	1.56	14.48	292	1266
	Core 2 Quad						
1	15.12	1.22	1.41	1.46	22.03	47.54	33.93
2	7.53	1.15	0.79	1.0	19.96	56.61	19.89
4	3.96	0.81	0.64	0.6	19.96	76.34	12.14

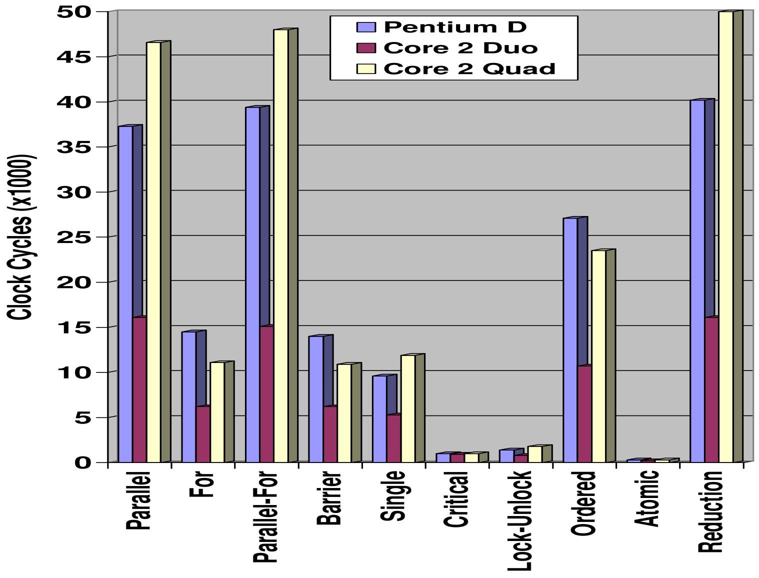

FIG. 6.1. *OpenMP Synchronization overheads of two threads on Intel Pentium D, Intel Core 2 Duo and Intel Core 2 Quad machines.*

Tables 4, 5 and 6 show the OpenMP loop scheduling overheads of the Static, Dynamic, and Guided clauses for chunk sizes of 1 to 128 when running 1, 2 and 4 threads, for Intel Pentium D, Intel Core 2 Duo and Intel Core 2 Quad machines respectively. Figures 3-5 are bar charts of the results for the case of two threads.

It can be observed that each clause has a different pattern. In the case of Intel Pentium D and two threads (Figure 3), the block cyclic scheduling (Static) presents similar overhead for all chunk sizes ($\sim$ 15K cycles). The Dynamic scheduling overhead decreases rapidly from the maximum point at a chunk size of one (92K cycles) to the minimum point at a chunk size of 128 (19K cycles). The Guided scheduling overhead has a similar pattern but with a more moderate decreasing curve. The maximum point is at a chunk size of one (92K cycles) and the minimum point is at a chunk size of 128 (14K cycles). The conclusion is that by increasing the chunk size the loop scheduling overhead is minimized.

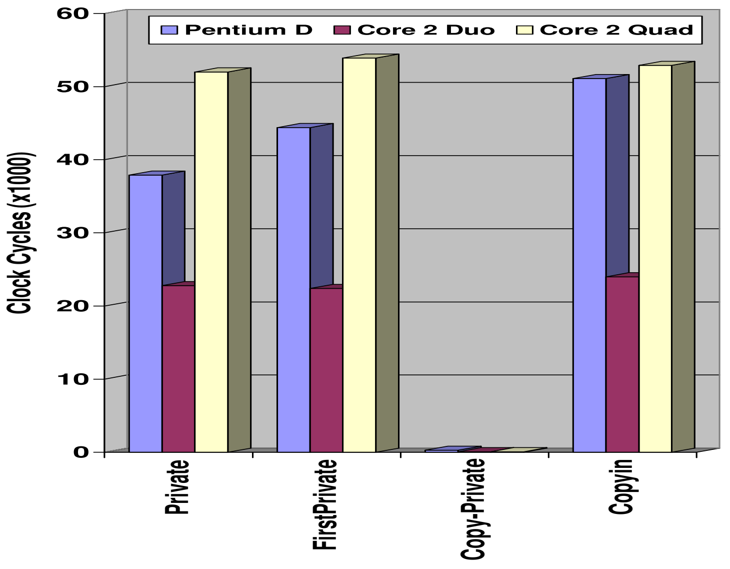

FIG. 6.2. *OpenMP Array Scheduling (privatization) overheads of two threads on Intel Pentium D, Intel Core 2 Duo and Intel Core 2 Quad machines.*

In the case of Intel Core 2 Duo and two threads (Figure 4), the loop scheduling overheads present similar pattern compared to the Intel Pentium D, but the overheads are up to 50% lower than those of the Intel Pentium D. The block cyclic scheduling (Static) is reach its optimal overhead at a chunk size of 64 (6K cycles). The Dynamic scheduling overhead decreases rapidly from the maximum point at a chunk size of one (48K cycles) to the minimum point at a chunk size of 64 (8K cycles). The Guided scheduling overhead has a similar pattern but with a more moderate decreasing curve. The maximum point is at a chunk size of one (10K cycles) and the minimum point is at a chunk size of 128(7K cycles).

In the case of the Intel Core 2 Quad and two threads (Figure 5), the loop scheduling overheads present similar pattern compared to the Intel Pentium D, but the overheads are more fluctuated. The block cyclic scheduling (Static) is reach its optimal overhead at a chunk size of 16 (7K cycles). The Dynamic scheduling overhead decreases rapidly from the maximum point at a chunk size of one (88K cycles) to the minimum point at a chunk size of 16 (10K cycles). The Guided scheduling overhead has a similar pattern but with a more moderate decreasing curve. The maximum point is at a chunk size of one (17K cycles) and the minimum point is at a chunk size of 32(8K cycles). It can be observed again that in the case of the Intel Core 2 Quad and four threads (Table 6), the scheduling overhead is up to twice compared to the case of two threads but still reasonably, in contrast to the scheduling overhead associate with four threads on the Intel Pentium D and the Intel 2 Core Duo where the overhead costs, due to less cores, are unacceptable (Table 4 and 5).

The bottom line of the EPCC micro-benchmarks results is that the overhead incurred by the OpenMP directives and the clauses are low and will not harm the performance of the NBP application benchmarks.

The NPB benchmarks were conducted with three different input sizes: S, W, and A for 1, 2 and 4 threads. The total running time of each benchmark was measured by wall-clock time. The speedup, efficiency, and the overhead of each run were calculated as follows:

Speedup $= T_1/T_{p,k}$ where T_1 is the time measured for running with a single core and $T_{p,k}$ the time measured with p cores and k threads.

Efficiency $= T_1/(p \cdot T_p)$ where T_1 is the time measured for running with a single core and T_p the time measured with p cores.

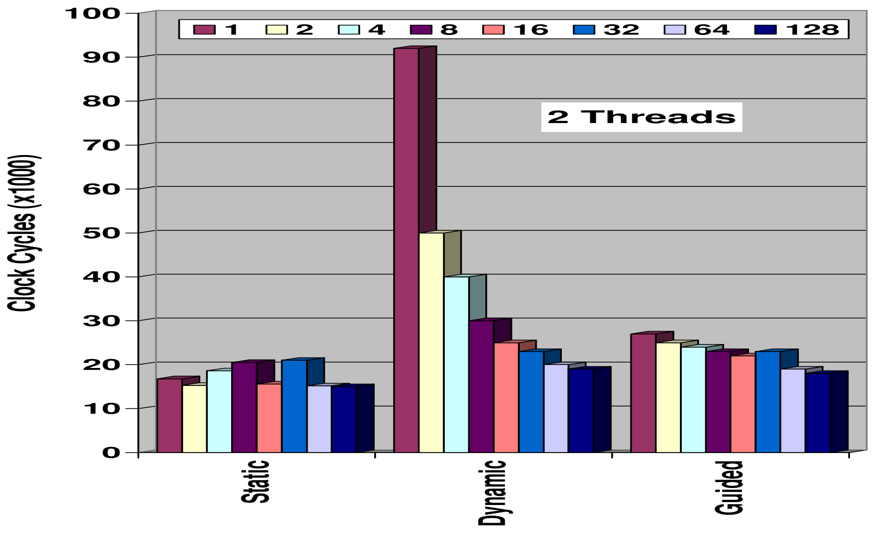

FIG. 6.3. *OpenMP Loop Scheduling overheads of two threads on Intel Pentium D machine.*

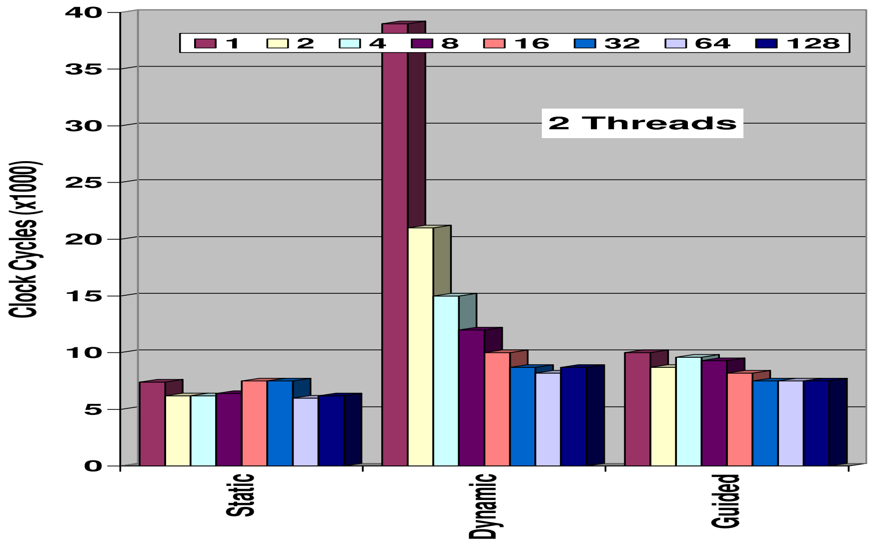

FIG. 6.4. *OpenMP Loop Scheduling overheads of two threads on Intel Core 2 Duo machine.*

$Overhead = T_{p,k} - T_1/p$ where T_1 is the time measured for running with a single core and $T_{p,k}$ the time measured with p cores and k threads.

We ran the original benchmarks that appear in [6] without any modification. Table 7 presents the performance (in Giga clock cycles) of class W (24x24x24) of seven different benchmarks in the NAS OpenMP suite for 1, 2 and 4 threads on the tested machines. Figure 6 is a bar chart of the calculated speedup of the NPB benchmarks, for the case of two threads, on the Intel Pentium D, the Intel Core 2 Duo and the Intel Core 2

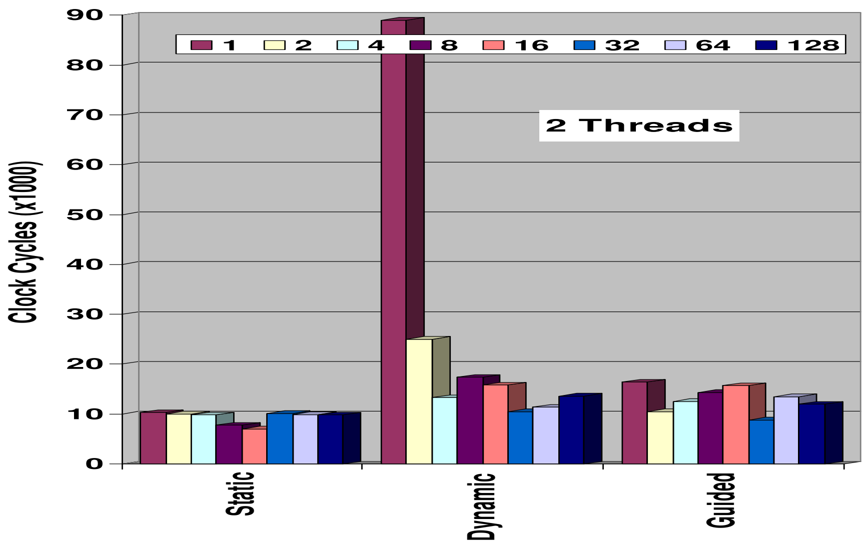

Fig. 6.5. *OpenMP Loop Scheduling overheads of two threads on Intel Core 2 Quad machine.*

Quad machines. However, the behavior of the benchmarks results for input classes S and A are similar to the input class W. The results of the IS kernel are omitted due to a problem known to the developers of the benchmarks suite , which they will fix in the next release. Figure 7 is a bar graph that depicts the percentage of the computation time and the overhead time as part of the total execution time, in the case of the Intel Pentium D machine.

Analysis of these results leads to the following findings.

The EP Kernel falls into the category of applications termed “embarrassingly parallel” based on the trivial partition ability of the problem, while incurring no data or functional dependencies, and requiring little or no communication between processors. It is included in the NPB suite to establish the reference point for peak performance on a given platform. Therefore, it is not surprising that the EP kernel achieved perfect speedup ($\sim$ 2.0) for two threads.

As can be observed from Figure 6, the speedups of the Intel Pentium D and the Intel Core 2 Duo processors are similar, with slight improvement in the case of Intel Core 2 Quad processor. The LU decomposition application, in case of Intel Pentium D, shows good speedup and efficiency for two cores, 1.77 and 0.88 respectively, but in the case of four threads the speedup drops drastically to 0.01 due to fewer cores than threads. The Intel Core 2 Duo processor achieves speedup of 1.71 and efficiency of 0.85 for two threads but drops to speedup of 0.02 for the case of four threads. On the other hand, the Intel Core 2 Quad achieves speedup 1.7 (0.85 efficiency) and 2.73 (0.68 efficiency) for two and four threads respectively (Table 7). The MG kernel shows more modest speedups (1.49 and 1.51) and efficiency (0.74 and 0.75) for two threads while for four threads the speedups drop to 0.89 and 0.79 on the Intel Pentium D and the Intel Core 2 Duo respectively. In the case of Intel Core 2 Quad the speedups are 1.78 (0.89 efficiency) and 2.20 (0.55 efficiency) for two and four threads respectively.

The rest of the benchmarks (FT, CG, BT, and SP) show poor speedups and efficiencies. These results can be explained by the logical structure of the applications, which do not match the underlying architectures of the multicore processors. For example, the FT kernel uses FFT on a complex array to solve a three-dimensional partial differential equation. Communication patterns in this kernel are structured and long distance in nature. This benchmark represents the essence of many “spectral” codes or eddy turbulence simulations. The CG kernel is used in conjugate gradient methods to approximate the smallest eigen-value of a symmetric, positive definite, sparse matrix with a random pattern of non-zeros. The communication patterns in this kernel are long- distance and unstructured.

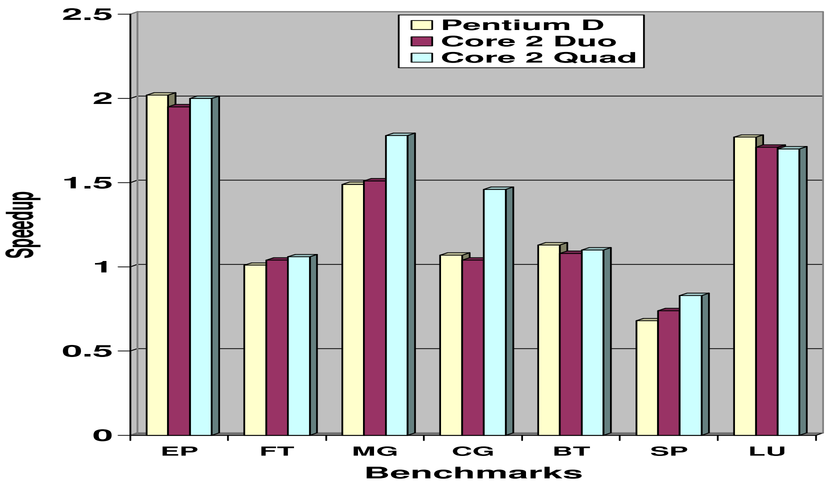

FIG. 6.6. *NPB speedup results of EP, FT, MG, CG, BT, SP, and LU benchmarks for 2 threads; problem class W; on Pentium D, Core 2 Duo and Core 2 Quad machines.*

These observations reveal the following conclusions. First, all the benchmarks, except EP, achieve very poor efficiency when the number of threads (4) is greater than the number of cores (2) which occurs in the cases of Intel Pentium D and Intel Core 2 Duo processors. In the case of Intel Core 2 Quad processor and four threads there is improvement in the speedups when changing the number of threads from two to four threads, except to the SP benchmark that exhibits poor performance in all the cases. It happens because the overhead caused by context-switch operations of the competing threads on the CPU resources is high. Moreover, two threads sharing a single core lead to cache conflicts that decrease the hit rate and thus degrade the performance.

The above poor performance of the NPB applications brought us, on the one hand, to extend our study further and to look for possible solutions to improve the performance but without restructuring the application programs, and on the other hand, to extend our understanding of how the underlying hardware works.

The Intel Pentium D processor has a different cache-memory architecture than Intel Core 2 Duo and Intel Core 2 Quad processors [8, 9]. The Pentium D 820 processor is a "distributed" cache-memory with two separately L1 caches of 16KB each and two separately L2 caches of 1MB each. On the other side, the Intel Core 2 Duo E6300 is a "shared" cache-memory with two separate L1 caches of 32KB each and a shared L2 cache of 4MB and the Intel Core 2 Quad Q6600 has two separate pairs of dual-core with 4MB shared L2 cache, and 2x32KB L1 caches. To understand the implications of these three different architectures on the performance, lets look at the following example.

Let A[100] be a shared array used by two threads running on two different cores. The threads are writing the array at the same time. One thread accesses the first part of the array, A[0-49], and the second one accesses the second half of the array, A[50-99]. In the case of the Intel Pentium D, array A will be copied into the L1 and the L2 caches of each core. Now, each time one of the threads completes a write operation, there is a need to update the copy of the array A in the neighboring core in order to maintain the caches consistency. This update is costly in terms of CPU cycles, as can be seen in Table 8. However, we expected to an improvement in the case of Intel Core 2 Duo processor because the L2 cache is shared, but we were disappointed to discover that the speedups of NBP benchmarks showed only $\sim$ 4% improvment (in the case of FT, MG and SP) compared to the Intel Pentium D, and $\sim$ 4% worsening (in the case of CG, BT and LU) when running with two threads.

So, we continued to explore further and we found another obstacle that we were not aware of: the shared L2 cache of the Intel Core 2 Duo is not banked. The L2 cache serves only one core at any given clock cycle, so a banked organization will not help. A round robin scheme is used to allocate L2 cache services to the cores for

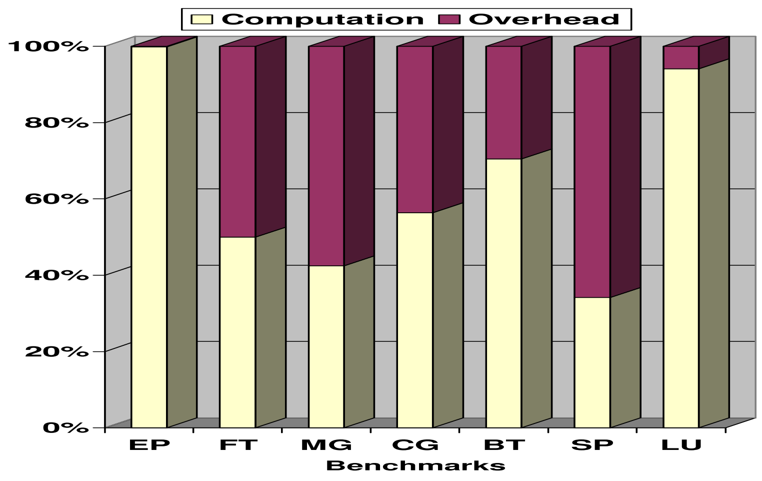

Fig. 6.7. *NPB computation time vs. overhead time of EP, FT, MG, CG, BT, SP, and LU benchmarks for 2 threads and class W on Pentium D machine.*

Table 6.7
False sharing penalties.

Case	Data Location	Latency(cycles/nsec)
L1 to L1 Cache	L1	14 core + 5.5 bus
Through L2	L2	14 core
Through Memory	Main Memory	14 core + 5.5 bus + 40-80 nsec

scenarios when both cores request L2 service. The false sharing penalty of the Intel Core 2 Duo is depicted in Table 8 and was taken from [9].

We looked further for optimization possibilities for improving the performance of the applications but without need to rewritten the programs.

First, we used a thread affinity option to tie a thread to its data to improve data locality [11]. Since OpenMP does not support thread affinity capabilities we used the Windows operating systems *SetThreadAffinityMask* option. Monitoring the threads scheduling by the Intel VTune performance analyzer confirmed that each thread was tied to one core during the program execution. Unfortunately, we did not observe any improvement in the performance of the applications. Second, we changed the loop iterations scheduling by using the OpenMP schedule clause. The fact that most of the parallelism of the NPB applications is done by *for* work-sharing, encouraged us to find the optimal scheduling. We tried the Static, Dynamic, and Guided options with 1, 4, 8, 16, 32, 64, and 128 chunk sizes. Unfortunately, we cannot report any significant improvement in the applications performance.

To summarize this section we list the key observations of our study.

- The impact of the parallel overheads of the OpenMP parallel mechanisms on the performance of NPB benchmarks is negligible.
- The new Intel multicore processors reduce the parallel overheads incurred by the OpenMP mechanisms compared to their predecessors.
- It is recommended to turn off the OpenMP compiler option when running single threaded applications because the relatively high overhead incurred by OpenMP.

- It is recommended to use one thread per processor-core to prevent the competition of two threads on a single core.
- Using the OpenMP directives Parallel and For together rather than separately incur less overhead.
- It is recommended to use the low-overhead OpenMP Atomic directive, where it is possible, instead of other locking mechanisms.
- OpenMP Loop scheduling incurs less overhead when the chunk size increases.
- The poor performance of five NPB benchmarks (FT, MG, CG, BT, and SP) is due to lack of matching between the logical structures of the applications and the underlying architecture of the multicore processors.

7. Conclusions. Multicore processors will dominate scientific computing, and commercial computing as well, in the near future. Understanding their performance characteristics is essential for design scalable and efficient applications. In this paper, we presented the efficiency of applications from NPB OpenMP-C suite and the overhead measurements of OpenMP directives and clauses running on Intel Pentium D, Intel Core 2 Duo and Intel Core 2 Quad machines using MS Visual studio C++ 2005 and Intel C++ compilers.

The benchmarking results show that most of the applications achieved poor performance, not because of the overhead incurred by the OpenMP directives, but because the NPB applications induced computation and communication patterns which are not cache friendly and result in a lot of false sharing situations. The diversified cache architectures of multicore processors call for new parallel programming languages and compilers that can use the hierarchy of cache memory systems in an efficient manner.

REFERENCES

[1] Geer D. (2005), "Chip makers turn to multicore processors," IEEE Computer.

[2] Marowka A. (2007), "Parallel Computing on Any Desktop," Communication of ACM, Vol. 50, Issue 9, pp. 74-78.

[3] "OpenMP Application Program Interface," `http://www.openmp.org.`

[4] Bailey D. H., Harsis T., Saphir W., Wijngaart R. V., Woo A., and Yarrow M., (1995) "The NAS Parallel Benchmarks 2.0," Report NAS-95-020, Nasa Ames Research Center.

[5] Jin H., Frumkin M., and Yan J., (1999) "The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance," Report NAS-99-011, Nasa Ames Research Center.

[6] "The Omni Project," `http://phase.hpcc.jp/Omni/home.html`

[7] Bull M. (1999), "Measuring Synchronisation and Scheduling Overheads in OpenMP," Proceeding of First European Workshop on OpenMP (EWOMP '99) Lund, Sweden.

[8] Doweck J., (2006) "Inside Intel® Core^{TM} Micro architecture and Smart Memory Access," A White Paper, Intel.

[9] Mendelson A., Mandelblat J., Gochman S., Shemer A., Chabukswar R., Niemeyer E. and Kumar A., (2006) "ICMP Implementation in Systems Based on the Intel® Core^{TM} Duo Processor," Intel® Technology Journal, Vol. 10, Issue 02.

[10] Furlinger K., Gerndt M., and Dongarra J., (2007) "Scalability Analysis of the SPEC OpenMP Benchmarks on Large-Scale Shared Memory Multiprocessors," Proceeding of ICCS.

[11] Tian T., (2007) "Tips for effective usage of the shared cache in multicore architectures," Embedded magazine, `http://embedded.com/showArticle.jhtml?articleID=196902691`

[12] Bull M. and O'Neill D., (2001) "Microbenchmark Suite for OpenMP 2.0," Proceedings of the Third European Workshop on OpenMP (EWOMP'01), Pages: 41–48, Barcelona, Spain.

[13] "Sphinx Micro-benchmark Suite," `http://www.llnl.gov/CASC/RTS\_Report/sphinx.html`

[14] Marowka A., (2008) "Performance of OpenMP Benchmarks on Multicore Processors," 8th International Conference on Algorithms and Architectures for Parallel Processing(ICA3PP), Agia Napa, Cyprus, LNCS proceeding Vol. 5022 pp. 208-219.

[15] Liao C., Liu Z., Huang L., and Chapman B. (2005) "Evaluating OpenMP on Chip MultiThreading Platforms," Proceeding of first international workshop on OpenMP (IWOMP 2005), Eugene, Oregon USA.

[16] Milfeld K., Goto K., Purkayastha A., Guiang C. and Schulz K. (2007) "Effective Use of Multi-Core Commodity Systems," The 8th LCI International Conference on High-Performance Clustered Computing, Lake Tahoe, California, 2007.

[17] Furlinger K., Gerndt M. and Jack Dongarra J. (2007) "Scalability Analysis of the SPEC OpenMP Benchmarks on Large-Scale Shared Memory Multiprocessors," Proceeding of ICCS, LNCS Volume 4488, pp. 815–822.

[18] Grant E. R. and Afsahi A. (2007) "A Comprehensive Analysis of OpenMP Applications on Dual-Core Intel Xeon SMPs," IEEE Proceeding of Parallel and Distributed Processing Symposium (IPDPS 2007).

[19] Curtis-Maury M., Ding X., Antonopoulos C. D. and Nikolopoulos D. S. (2008) "An Evaluation of OpenMP on Current and Emerging Multithreaded/Multicore Processors," M. S. Mueller et al. (Eds.): IWOMP 2005/2006, LNCS 4315, pp. 133–144, 2008.

Edited by: Marcin Paprzycki
Received: May 25th, 2009
Accepted: November 25th, 2009