# Computer Science Technical Report

## ScaleUPC: A UPC Compiler for Multi-Core Systems

by Weiming Zhao, Zhenlin Wang

Michigan Technological University
Computer Science Technical Report
CS-TR-08-02
September 8, 2008


Department of Computer Science
Houghton, MI 49931-1295
www.cs.mtu.edu

# ScaleUPC: A UPC Compiler for Multi-Core Systems

Weiming Zhao, Zhenlin Wang

September 8, 2008

**Abstract**

As multi-core computers begin to dominate the market, the lack of a well-accepted parallel programming language and its compilation support targeting this specific architecture has become an immediate problem to solve. Unified Parallel C (UPC), a parallel extension to ANSI C, is gaining great interest from the high performance computing community. UPC was originally designed for large scale parallel computers and cluster environments. However, the partitioned global address space programming model of UPC makes it a natural choice for a single multi-core machine, where the main memory is physically shared. This paper builds a case for UPC as a feasible language for multi-core programming by providing an optimizing compiler, called ScaleUPC, that outperforms other UPC compilers targeting SMPs.

Since the communication cost for remote accesses is removed because all accesses are physically local in a multi-core, we find that the overhead of pointer arithmetic on shared data accesses becomes a prominent bottleneck. The reason is that directly mapping the UPC logical memory layout to physical memory, as used in most of the existing UPC compilers, incurs prohibitive address calculation overhead. This paper presents an alternative memory layout, which effectively eliminates the overhead without sacrificing the UPC memory semantics. Our research also reveals that the compiler for multi-core systems needs to pay special attention to the memory system. We demonstrate how the compiler can enforce static process/thread binding to improve cache performance.

## 1 Introduction

With the advance in manufacturing technologies and concerns of physical limitations of microelectronics, multi-core computing becomes an inevitable trend in the computer industry. A natural use of a multi-core computer is to explore thread level parallelism (TLP) to speed up an application, requiring the application be multi-threaded. Current multi-threaded programming often uses a conventional imperative language plus the *pthread* library, which is notorious for its high complexity, low productivity, and poor portability. The high performance computing community has long sought a high-level parallel programming language which offers high performance as well as high productivity. OpenMP as an industrial standard for SMPs adds parallel directives to C/C++ and Fortran. However, the directives in OpenMP only serve as hints for parallelization that should instead be an integral part of the host language [1]. To this end, the languages with the partitioned global address space (PGAS) have shown advantages in large-scale parallel computers and cluster environments [2, 3]. Two representative PGAS languages are co-array Fortran and UPC.

UPC, a parallel extension to C, delivers high programmer productivity via its shared memory model, allowing programmers to be less concerned with low-level synchronization and communication across processors. In addition, UPC offers high performance via a partitioned memory space that provides effective parallelism and the inheritance of a well-developed software base from C. However UPC was designed for large-scale parallel computers or clusters. The compilation implementation often relies on MPI to implement the communication and parallelism. In a single multi-core machine, where the memory is both logically and physically shared, remote accesses and communication are no longer necessary. We propose a UPC compiler *ScaleUPC*, which directly translates a UPC program to a C program with pthreads.

For a UPC compiler under a cluster environment, communication and UPC pointer arithmetic are two key performance considerations. While C is already difficult to optimize due to pointers and aliases, UPC

brings additional complexity by differentiating pointers pointing to shared objects (pointers-to-shared) from pointers pointing to private objects (pointers-to-private). Dereferencing a pointer often requires complicated calculations and even run-time function calls. An optimizing compiler can reduce this complexity by casting a pointer-to-shared to a local pointer, called *privatization*. A UPC compiler can reduce communication cost through software caching, remote prefetching, and message coalescing [4, 5]. Our study shows that UPC pointer-to-shared arithmetic remains a significant part of overall execution time in a multi-core system. The pointer arithmetic overhead comes from the process of mapping the UPC global shared address space to the physical memory space. This paper presents a novel memory layout for shared objects, which can significantly reduce the extra computation cost and still maintain the compatibility to the UPC shared memory semantics. By using this new memory layout, ScaleUPC outperforms two peer UPC compilers, Berkley UPC [6] and Intrepid UPC [7], which can compile UPC for SMPs.

It was our original intent to design a compiler for a high level programming language such that the compiler can statically analyze parallel application's memory behavior and optimize for the multi-core memory system performance. This paper demonstrates the importance of memory system optimization by proposing a profile-driven static process binding scheme. Our compiler can statically bind a thread or process to a specific core. We show that there is up to a 66% performance gap among several simple binding policies and the compiler can help to find an optimal policy with the assistance of profiling.

# 2 Background and Related Work

## 2.1 Unified Parallel C

This section briefly introduces UPC. We focus on the parallel language features to which our compiler pays attention.

**Execution Model** UPC adopts the single program multiple data (SPMD) execution model. Each execution unit executes an identical program, denoted as a *UPC thread*. Each UPC thread can identify itself by an identifier *MYTHREAD*. UPC compilers typically implement a UPC thread as a process. Our compiler instead implements it as a pthread for efficient multi-core execution.

**Partitioned Shared Memory** The PGAS model of UPC gives programmers an illusion of logically shared memory space. Data objects in a UPC program can be either *private* or *shared*. A UPC thread has exclusive access to the private objects that reside in its private memory. A thread also has accesses to all of the objects in the shared memory. UPC partitions the global (shared) address space through the concept of *affinity*. The entire global address space is equally partitioned among all threads. The block of the global address space associated with a thread is said to have *affinity* to that thread. The concept of affinity captures the reality that on most modern parallel architectures the latencies of accessing different shared objects are different. It is assumed that for a shared object, an access from a thread that affiliates with that object is much faster than accesses from other threads. However, in a multi-core machine, this assumption no longer holds and affinity may be treated logically. Affinity does have impact on the performance of memory hierarchy, depending on the implementation.

The distribution of shared data objects among threads can be determined by specifying the *block size*, the number of objects in a block. The data in the same block are physically contiguous and have the same affinity and the blocks will be distributed to each UPC thread in a round-robin fashion. The offset of an object within a block is called a *phase*.

Based on the location of the pointed-to objects, there are two types of pointers in UPC: pointer-to-private and pointer-to-shared. The former one points to a private object and its arithmetic follows the same semantics as a pointer in ANSI C. A pointer-to-shared targets a shared object. The pointer arithmetic of a pointer-to-shared requires the knowledge of the type of the shared object and its attributes: affinity, block, and phase. To reduce the complexity of pointer-to-shared arithmetic, privatization can be applied by casting a pointer-to-shared to a pointer-to-private with certain transformations. In a multi-core environment,

privatization can still have a notable impact on overall application performance, depending upon the memory layout chosen by the compiler as discussed in Sections 3 and 5.

### 2.2 Related Work

There are quite a few UPC compilers available in academia and industry. However, to our knowledge, none of them directly targets multi-core systems. Some of them may compile UPC to run on a multi-core machine, but they use slower communication mechanisms such as message passing or implement a UPC thread as a process, which is more expensive in terms of context switching and inter-process communication than the kernel thread implementation.

UPC is part of the Cray C compiler for the Cray X1 [8]. A development version of UPC has been used for performance measurement on the IBM BlueGene/L at Lawrence Livermore National Laboratories [2]. Michigan Tech has developed a public domain UPC compiler with a run-time system that uses MPI as the transportation layer [9, 10].

The most widely used public domain UPC compiler is Berkeley UPC [6]. It has a highly portable run-time system because it provides a multi-layered system design that interposes the GASNet communication layer between the run-time system and the network [11]. GASNet works with various types of network, such as Myrinet, Quadrics, and even MPI. Berkeley UPC includes a UPC-to-C translator based on the Open64 open source compiler. Various optimizations, mainly for generating shared memory latency tolerant code, are done at the UPC source code level. Translator-level optimizations for Berkeley UPC are described in [12]. Though Berkeley UPC includes a SMP conduit and utilizes pthreads as an optimization for SMP systems, it suffers from the overhead of address arithmetic.

Intrepid Technology provides a UPC compiler [7] as an extension to the GNU GCC compiler. Intrepid implements a UPC thread as a process when running on Intel SMPs and uses memory map (mmap) as the mechanism of inter-process communication. Hewlett-Packard offered the first commercially available UPC compiler [13]. The current version of this compiler targets Tru64 UNIX, HP-UX, and XC Linux clusters.

In the area of process scheduling on multi-core, cache-fair algorithms [14] and cache partitioning [15] address the fair cache allocation problem. Parekh et al. [16] propose algorithms to identify the best set of workloads to run together. Nakajima and Pallipadi use hardware event counters to guide the scheduling [17]. Hardware solutions on shared cache management have also been proposed [18, 19]. All of them aim at scheduling multiple standalone workloads on shared resources. Our solution explores inter-thread locality of multi-threaded applications and focuses on identifying a specific process binding policy for SPMD programs to achieve better cache performance without hardware or OS-wide modification.

## 3 Compiler Design

In this section, we present the details of the construction of our UPC-to-C compiler.

### 3.1 Extensions to Scale Research Compiler

Our UPC compiler, ScaleUPC, is based on Scale (A Scalable Compiler for Analytical Experiments) [20], a compiler infrastructure developed by the University of Massachusetts, Amherst and the University of Texas, Austin. Scale provides C and Fortran frontends, a middle-end with most conventional compiler optimizations, and SPARC, Alpha, and Trips backends.

The Scale frontend transforms C or Fortran source code to an abstract syntax tree, called *Clef*. We extend Scale's C frontend and Clef-to-C module to translate a UPC program to a multi-threaded C program, where each pthread corresponds to a UPC thread.

### 3.2 Shared, Global, and Private Data Objects

In ANSI C, data objects declared in the file scope are both global to all routines and shared by all threads. In UPC, data declared in the file scope without a *shared* qualifier are still accessible to all routines, but each thread maintains a separate instance. We call them *global private (non-shared)* data.

In our implementation, local private data are treated as regular C local data, allocated on the stack. Shared data are translated into C global variables and thus are accessible to all threads. Global private data are allocated in thread-local storage (TLS) by specifying the *__thread*, a reserved keyword of GCC.

To reference the local private data and global private data, we can use regular C pointers. To reference a shared data object, we use the following 5-tuple to describe a pointer-to-shared in ScaleUPC:

1. *block address*: the physical address of the beginning of a block that the pointed-to object resides in;
2. *thread*: the thread affinity of the pointed-to object;
3. *phase*: the offset within the block;
4. *block number*: the logical block number (This field is not indispensable, but it facilitates the calculation of the logical distance from one pointer-to-shared to the other);
5. *type*: the type of the object that is pointed to, including the block size specified in the declaration or dynamic allocation, the size of each element, and the number of elements in the static array or allocated memory.

The physical memory address can be calculated as *block address* + *phase* $\times$ *element size*. However, all five fields except *type* need to be updated when pointers are manipulated. Depending on the underlying memory layout used for shared data, there is significant variance in terms of the complexity and cost of the pointer updating algorithm, which will be detailed in the next section.

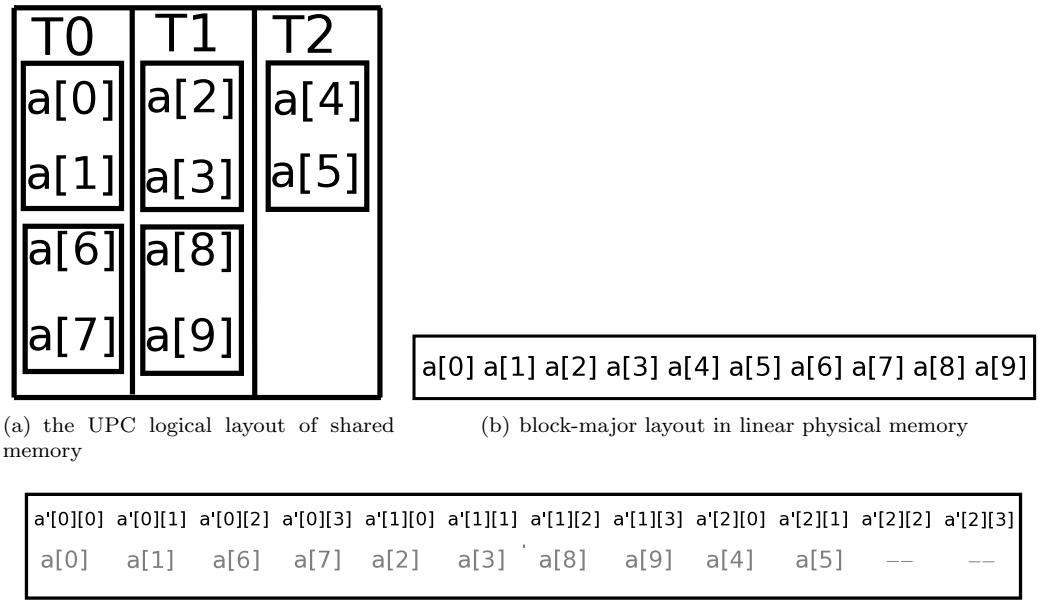
### 3.3 Memory Layouts for Shared Data

This section discusses two alternative memory layouts for UPC shared data and their impacts on array references and pointer-to-shared arithmetic.

We begin with an illustration, as shown in Figure 1(a), of a shared array $a$ of size 10, which is distributed across three threads with block size 2. By the UPC semantics, thread 0 has two blocks: one contains $a[0]$ and $a[1]$; the other $a[6]$ and $a[7]$, similarly for threads 1 and 2. If a pointer-to-shared pointer, $p$, points to $a[1]$, the increment of $p$ by 1 makes it point to $a[2]$. However, if $p$ is cast to a local pointer $q$ through privatization, the increment of $q$ by 1 makes it point to $a[6]$.

We implement two memory layouts for a shared array: *thread-major* and *block-major*. Thread-major follows a typical translation of a UPC compiler for a distributed shared memory system. The shared data with the same affinity are assembled as a one-dimensional array such that all local objects are next to each other physically. In our compiler, we translate array declaration $a[10]$ to $a'[3][4]$ where the first subscript denotes the thread affinity as shown in Figure 1(c). For example, under thread-major, $a'[1][2]$ refers to $a[8]$.

Thread-major adds complexity to the arithmetic of pointer-to-shared and shared array address calculations while privatization can simplify it. When a pointer-to-shared is cast (privatized) to a pointer-to-private, the thread-major layout would facilitate pointer arithmetic of the cast pointer, which has the same semantics as a regular C pointer since the local shared objects are physically mapped together. However, the translation of pointer arithmetic for a pointer-to-shared is much more expensive than the regular C pointer arithmetic. Note that $p + i$ in C is interpreted as the address in pointer $p$ plus $i$ multiplied by the size of the type that $p$ points to. It only requires two arithmetic operations in C. In UPC under the thread-major layout with a pointer-to-shared $p$, $p + i$ can cross a block and/or thread boundary. A correct calculation would involve updating the first four fields of a pointer-to-shared structure that requires at least 15 basic operations in our implementation. The large number of operations also implies a large number of temporaries and possibly




(a) the UPC logical layout of shared memory

(b) block-major layout in linear physical memory

(c) thread-major layout in linear physical memory

Figure 1: Logical and physical layouts of shared [2] int a[10], THREADS = 3

more register spills. When blocks are not evenly distributed among threads, two more basic operations are needed to adjust the block address.

For a numeric application where arrays are frequently used, the cost of shared array references is prohibitive due to the overhead of pointer-to-shared arithmetic since an array access like $a[i]$ virtually implies a pointer arithmetic of $a + i$. However, being different from a generic pointer-to-shared, the phase and thread affinity of $a$ are constantly 0. Taking advantage of this property, as long as the compiler can identify that $a$ points to the first element of a statically or dynamically allocated array (usually $a$ is the array name itself), the compile-time constant folding helps to reduce the number of basic arithmetic operations from 15 to 8 in our implementation and the need for temporary intermediate variables is lowered as well.

To reduce the complexity of pointer-to-shared arithmetic, we introduce an alternative layout, the *block-major* layout. The block-major layout keeps the original declaration as it is in the generated code, so $a[i]$ will be physically adjacent to $a[i + 1]$ as shown in Figure 1(b). This layout greatly simplifies the pointer arithmetic for a pointer-to-shared and makes privatization a redundant optimization.

When the block major layout is applied, pointer-to-shared arithmetic can have the same cost as regular C pointers. Although for pointers-to-shared the attributes such as thread affinity and phase still need to be maintained since they may be used by some UPC-specific operations such as *upc_threadof()* and *upc_phaseof()*, they can be computed with less operations than the thread-major layout. For a shared array reference, such as $a[i]$, its address calculation is the same as a regular C reference. We do not calculate any pointer-to-shared attributes unless they are needed. For instance, when the address of $a[i]$ is taken, a pointer-to-shared structure will be created by the compiler for future pointer references. By leaving a shared array reference in the same form as it is in the source code, a native C compiler can enjoy simpler dependence testing and possibly perform more dependence-related optimizations.

Though it is unnecessary to perform pointer privatization on a single multi-core computer with block-major, a pointer-to-shared needs to be casted to a pointer-to-private when passed as a parameter to native C/Fortran routines which assume the pointer being passed points to contiguous memory. ScaleUPC can detect this case, create a temporary buffer to hold the thread's local shared data, pass the buffer pointer to the external routines, and copy the data back after the call. This process can be viewed as a reverse process of privatization and seldom occurs in the benchmarks we use.

# 4 Profiling-Based Process Scheduling

UPC was designed for a cluster environment where communication cost is a major concern. To achieve high performance in a single multi-core machine, a UPC compiler must instead consider the impact of the memory hierarchy. This section discusses how process scheduling affects memory system performance and proposes a preliminary profiling-driven static thread binding scheme that explores inter-thread locality. We target a SMP machine with multiple multi-core processors.

In a hybrid system of SMP, CMP and/or SMT, caches and memory bus interfaces can be either shared or private. For a multi-threaded application, scheduling of the threads by the operating system can significantly affect the memory system performance [21]. For the simplicity of discussion, we assume an application with two threads. When the two threads are scheduled on two cores with dedicated caches and memory bus interfaces, there is no bus contention between the two threads nor inter-thread cache conflict. However, this scheduling fails to explore inter-thread locality and also increases bus traffic and likely latency for maintaining cache coherence. On the other hand, when the two threads are scheduled on the same die or domain with a shared cache, the two threads will compete for cache and system bus, and may cause inter-thread conflicts. However, the scheduling has potential to improve the hit rate depending on the inter-thread locality.

By default, the thread scheduler in the current Linux kernel prefers to bind each thread to a core with dedicated resources, which may benefit some benchmarks but hurt the others [21]. Ideally, to achieve optimal performance for every workload, threads need to be scheduled dynamically based on their memory behavior. We propose a simple, profiling-based algorithm to determine the process/thread binding policy statically. The profiler compares the miss rates between the two scheduling policies for a training input. It suggests binding the two threads into two cores with a shared cache only when the profiler finds it would improve the cache hit rates over a threshold. The profiler then feeds back this hint to the compiler which inserts OS API calls to enforce the binding. Given two threads, we profile their miss rates assuming a shared cache and their miss rates under separate caches. We calculate the miss rate as the total L2 misses of the two threads divided by their total L2 accesses. Assume that the miss rates are $x$ and $y$ respectively for the two settings and also assume the cache hit latencies are $H_1$ and $H_2$ and miss latencies are $M_1$ and $M_2$. The average cache access latencies of one access are $(1-x)*H_1+x*M_1$ and $(1-y)*H_2+y*M_2$ respectively. Binding two threads to the same cache would benefit if $(1-x)*H_1+x*M_1 < (1-y)*H_2+y*M_2$ which is reduced to $\frac{x}{y} < \frac{M_2-H}{M_1-H}$ when $H_1 = H_2 = H$.

Due to the randomness of system scheduling during execution, the same program with static binding may show quite different access patterns in different runs. When two threads share the last level cache, an access to shared data may hit the cache in one run but miss it in the other. This happens when one thread makes more progress than the other during a period of execution due to some random factors such as context switches, interrupts, and varied memory latency. When the gap between the two threads is large enough, a cache line brought into cache by one thread might be evicted before it gets hit by the other. We call this *out-of-phase execution*. Section 5.4 demonstrates this phenomenon experimentally and suggests that additional forced synchronization by the compiler may improve performance.

# 5 Experimental Evaluation

In this section, we first present the effect of our new UPC shared memory layout and compare our compiler with Berkeley UPC, Intrepid UPC, and GNU OpenMP. We then show how the memory system influences the UPC application performance and thus future compiler design for multi-core systems. We further evaluate our profile-driven process binding optimization.

We use George Washington University's Matrix Multiplication (MM), N-Queen (NQ) and the UPC implementation of the NAS NPB benchmark suite [22]. MM computes $C = A \times B$ by a basic 3-loop algorithm ($O(n^3)$). We choose an array size of $2048 \times 2048$ for $A, B$ and $C$. We intentionally transpose $B$ for simple cache and locality analysis. Arrays $A, B$ and $C$ are all declared as shared arrays with a block size of 2048, denoting round-robin distribution of rows across UPC threads. For NQ, we use a $16 \times 16$ chess board. The UPC version of the NAS NPB benchmark suite includes CG, EP, FT, IS, and MG. Each benchmark has

a basic, untuned version (O0). The O0 set will be referred as NPB O0 later on [1]. All NPB benchmarks except EP have hand-optimized versions that implement privatization (O1) and/or remote prefetching (O2) [5] at the UPC source code level. Since the O2 version implies O1, it is called O3 instead, which means O1 plus O2. We choose the versions with the highest optimization option from the suite and refer to the set as NPB O3. It is expected that an optimizing compiler can match the performance of the O3 set by optimizing the O0 set. Each benchmark has a *class* parameter to specify the problem size. The parameter, in increasing order of problem size, can be S, W, A, B, C or D. In our experiments, we use class B, the maximum problem size that enables all NPB benchmarks to run in our test multi-core machine without exceeding the limit of its physical memory size.

We run the benchmarks on a Linux 2.6 based 8-core server, which is equipped with two Intel Xeon 5345 2.33 GHz Quad-Core processors and a 4 GB DRAM with 667 MHZ Dual Ranked Fully Buffered DIMMs. Each processor consists of two dual-core dies. The processor has 2-level caches: 32KB private L1 data cache per core and 4 MB shared L2 cache per die (8MB total cache per processor) [23]. The Memory Controller Hub on the server board supports an independent 1333 MHz front side bus interface for each processor [24]. Figure 2 illustrates the topology of the machine.

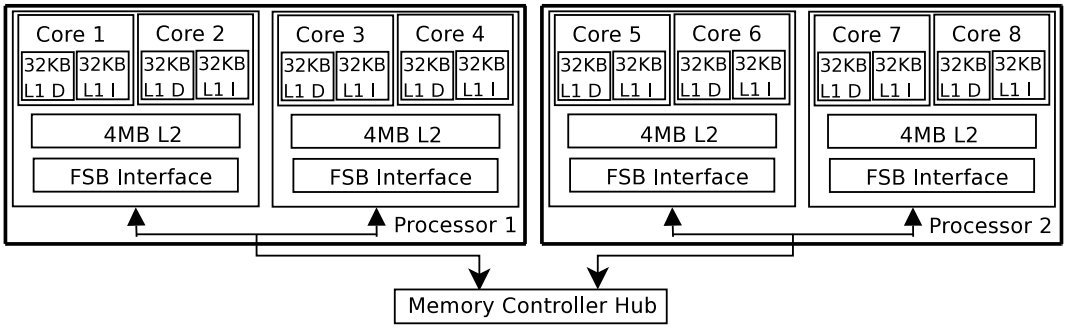



Figure 2: Topology of the 8 cores

## 5.1 Performance Comparison

In this section, we compare the performance of ScaleUPC with Intrepid UPC 4.0.3.4 and Berkeley UPC 2.6.0. We turn on -O3 and -O options, for Intrepid UPC and Berkeley UPC respectively, to generate optimized target code. For reference purposes, we also measure the performance of NPB's OpenMP implementations [25]. The NPB OpenMP benchmarks are compiled using GNU's OpenMP compiler 4.1.2. All OpenMP benchmarks except IS are hosted in Fortran while IS is in C. Since OpenMP and UPC are two different languages, the performance gap between them can help reveal the potential performance profit that the slower one can exploit.

When compiled by Berkeley UPC and Intrepid UPC, FT experiences exceptionally long execution time, and so does IS under Intrepid UPC. We exclude those outliers in our statistics. ScaleUPC is able to compile all benchmarks correctly.

### 5.1.1 Block Major vs. Thread Major

Figure 3 displays the normalized execution times when each benchmark is compiled to eight threads. The code generated by ScaleUPC using the block-major layout shows the best overall performance. The degree of improvement largely depends upon the frequency of pointer-to-shared arithmetic. NQ and EP are embarrassingly parallel and have few shared memory access, so the difference in performance for each compiler is small. For the programs with intensive shared memory accesses including MM and all NPB O0, except EP, block major layout has a significant advantage due to its simple pointer arithmetic. For MM,

[1] MM and NQ are untuned too. So in our analysis, we discuss them with the O0 group.

the block-major layout is 5.4 times faster as the thread-major layout. The block major layout brings an average of 3.16 speedup over the thread major layout for NPB O0. However, as shown in Figure 3(b), the gap between block-major and thread-major becomes barely noticeable for NPB O3 because privatization and remote prefetching essentially eliminate *shared* accesses.

Under the block-major layout, the difference in execution times between each O0 and O3 version is within 1% except IS whose O0 version is 4.8 times slower than the O3 version. The IS O0 source code uses a different way to store keys than the code in the O3 version, which involves extra calculations for every access to the keys. In other words, the IS O3 version not only applies privatization and remote prefetching, but also changes the data structure and algorithm. When we modified the IS O0 code to use the same algorithm and structure as used in the O3 set for fair comparison, the gap was reduced to a few percent. In general, when the block major layout is applied by ScaleUPC, the non-optimized O0 can deliver comparable performance to the hand-optimized O3 set. The thread major still needs privatization and remote prefetching to compete.

#### 5.1.2 ScaleUPC vs. Berkeley and Intrepid UPC

As shown in Figure 3(a), for MM, NQ, and NPB O0, ScaleUPC/block-major outperforms Berkeley UPC and Intrepid UPC. On average, ScaleUPC/block-major delivers a speedup of 2.24 and 2.25 over Berkeley UPC and Intrepid UPC, respectively. The performance gap is largely from the overhead of shared data layout as implied by comparing with ScaleUPC/thread-major and the performance of NPB O3. Both Intrepid UPC and Berkeley UPC maintain the memory layout in a thread-major like style. The average performance of Berkley UPC and Intrepid UPC is more comparable to that of ScaleUPC/thread-major for MM, NQ, and NPB O0: ScaleUPC/thread-major still gains 7% speedup over Intrepid UPC, but loses to Berkeley UPC by 30%.

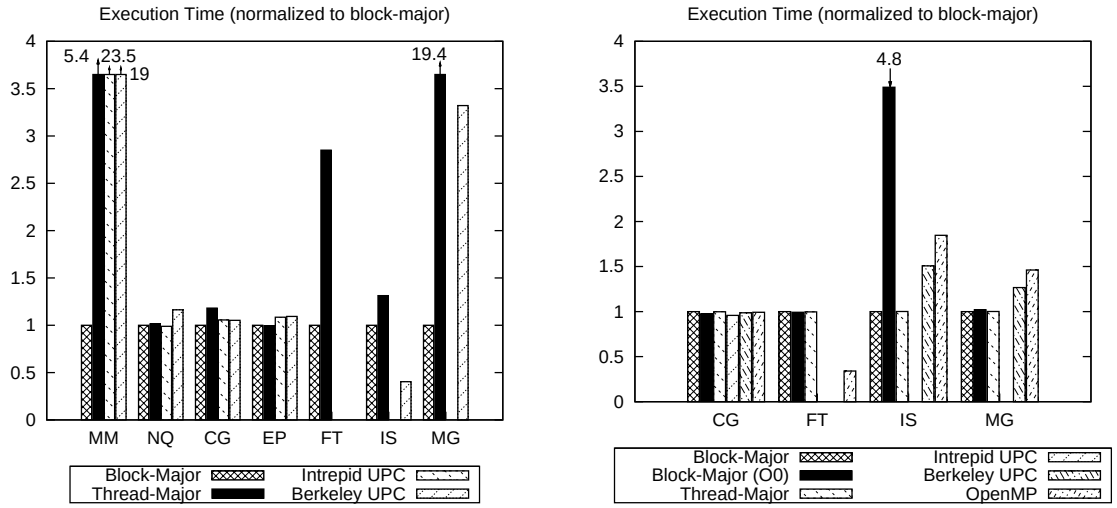
The performance gaps among all compilers are significantly reduced for the benchmark set of NPB O3 as shown in Figure 3(b). As described early, NPB O3 is a hand-optimized benchmark suite that applies privatization and remote prefetching so that the shared data accesses are transferred to local accesses. With these two optimizations, the benefit of block-major which reduces the cost of address arithmetic of shared data accesses is cut down to the minimum. As a result, ScaleUPC/block-major and ScaleUPC/thread-major show almost the same performance. Intrepid UPC outperforms Scale UPC by 4% for CG, the single benchmark in NPB O3 it can compile correctly. Berkeley UPC is still 23% behind ScaleUPC, on average, for CG, IS and MG. Based on the results in Figure 3, we conclude that privatization and remote prefetching are still two indispensable optimizations that a multi-core UPC compiler needs to implement if the thread-major layout is taken. However, the two optimizations are redundant when the block-major layout is applied at compile time.

#### 5.1.3 ScaleUPC vs. OpenMP

The UPC benchmarks compiled by ScaleUPC/block-major deliver comparable performance as their OpenMP counterparts compiled by GCC. On average, OpenMP gains merely 2% over ScaleUPC/block-major. It suggests that an application written in UPC can compete with an application written in OpenMP when the compiler design at the UPC side catches up. Taking both Figure 3(a) and Figure 3(b) into account, we can observe a significant gap between OpenMP and Intrepid UPC for NPB O0, both of which are GNU based compilers. It suggests that the multi-threading implementation and the block-major memory layout proposed in this paper are two key parts for UPC to achieve high performance in a multi-core environment. Considering the lost optimizing opportunities due to source to source compilation by ScaleUPC, UPC has potential to outperform OpenMP significantly in performance and can be an excellent choice of programming language for multi-core programming.

### 5.2 Scalability

This section analyzes the scalability of UPC programs compiled by ScaleUPC. We run the benchmarks on the Xeon 8-core server with 1, 2, 4 and 8 thread(s) respectively. In general, the scalability is limited by:




(a) MM, NQ and NPB O0 on Xeon / 8 threads

(b) NPB O3 on Xeon / 8 threads

Figure 3: Comparison on execution time

1) the intrinsic parallelism of programs; and 2) memory bus throughput. For the O0 set, as shown by Figures 4(a) and (b), NQ and EP exhibit perfect linear speedup since they are both embarrassingly parallel and computation bound programs. IS O0 shows a slight performance degradation when scaled to 2 and 4 threads under block-major. The data structure used in IS O0 does not scale for block major. Although IS O0 gains close to linear speedup under thread major, its performance is still far behind block major with a 31% slowdown at 8 threads, for instance. MM and FT experience a slight performance degradation from single-thread to 2-thread execution under the thread-major layout. This is because when the thread count is 1, most multiplication, division and modulo operations used for address calculation are removed at compile time. From 2 threads to 4 and 8 threads, the speedups of MM and FT come back. CG shows no speedup or even 1.5% slowdown from 4-thread to 8-thread for NPB O3 under both layouts and NPB O0 under block major layout. CG compiled by Intrepid UPC and Berkeley UPC does not scale from 4 to 8 threads either. We thus suspect CG cannot scale well beyond four threads.

For NPB O3, the block-major and thread-major layouts show almost the same speedups where the difference is below 1%. Also, the speedups of NPB O0 under block-major are close to those of NPB O3 under both memory layouts except for IS as we have explained in Section 5.1.1.

## 5.3 Effects of Profiling-Based Static Processor Scheduling

Section 5.1 shows that the block major memory layout can bring a speedup up to a factor of five. Although we attribute the speedup to simpler pointer arithmetic, we still believe the memory hierarchy would play a key role for the performance of multi-threaded applications and the compiler should take it into account for optimization. This section showcases the importance of the memory hierarchy by evaluating the effects of our profiling-based processor scheduling algorithm that we present in Section 4. ScaleUPC is able to exploit the memory system by applying appropriate binding schemes suggested by the profiler.

To profile the benchmarks, we develop a cache simulation tool based on the PIN tool set [26] to simulate the cache hierarchy of the Xeon Quad-Core processor. We profile the benchmarks using small inputs: *class A* for NPB O0 benchmarks, $1024 \times 1024$ arrays for MM and a $8 \times 8$ chess board for NQ. Each benchmark is compiled to two threads for profiling because two threads allow us to evaluate all possible processor bindings on the 2-way Quad-Core Xeon machine and we use block-major layout as it delivers the best performance compared to thread-major and other compilers.

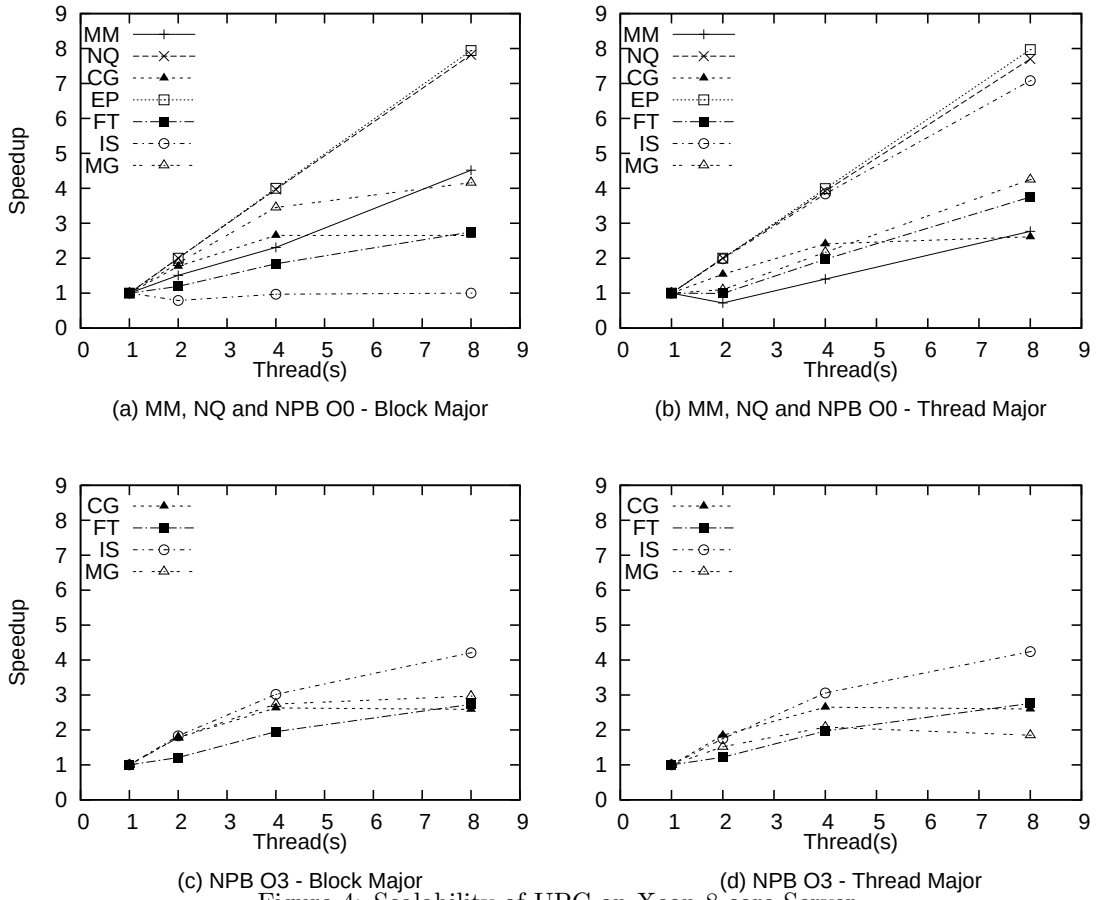



Figure 4: Scalability of UPC on Xeon 8-core Server

The second and third columns of Table 1 report the profiling results. The second column shows the miss rates in percentage when the two threads share a 4M L2 cache, emulating the scenario when the two threads are bound to the same die. The third column shows the miss rates when the two threads run on two separate 4M L2 caches, emulating the case when the two threads are bound to separate dies or processors. The fourth column shows the ratio of the two miss rates. Note that only MM and IS O0 show significant miss reduction when the two threads share L2 cache.

We measure the execution times of the benchmarks with larger inputs: *class B* for NPB benchmarks, $2048 \times 2048$ arrays for MM and a $16 \times 16$ chess board for NQ. The last three columns of Table 1 report the speedup under different process binding schemes over the default Linux scheduling. The *shared-L2* scheme binds two threads to the same die that shares the L2 cache. For the *same-package* scheme, two threads are bound to two separate dies that reside in same package, so each has full access to its own L2 cache. The *separate-package/processor* scheme sends two threads to two processors, which is most similar to the default Linux thread scheduling and gives each thread full access to the caches and memory bus interfaces.

Although, by geometric mean, the three binding schemes are all within 4% of the default scheduler, there are significant variances in the individual benchmarks. The largest gap is observed for IS O0 where the best scheme outperforms the worst by 66%. When two threads are bound to separate packages, the overall performance is slightly better than under Linux default scheduling because the OS will migrate each thread between cores within the package even though the current core would be idle. For our Quad-Core Xeon machine, when there are few shared-data accesses, *separate-package* should perform better than both *shared-L2* and *same-package* since two threads have separate caches and independent front side bus. *Shared-L2* will excel over *separate-package* only when inter-thread locality improves the hit rate to the degree that

| Benchmark | L2 Misses (%) | | $\frac{Sha.}{Sep.}$ | Speedup | | |
|---|---|---|---|---|---|---|
| | Sha. L2 | Sep. L2 | | Sep. Pac. | Same Pac. | Sha. L2 |
| MM | 50.0 | 99.9 | 0.50 | 1.06 | 0.90 | 1.21 |
| NQ | 0 | 0 | – | 1.00 | 1.00 | 1.00 |
| CG O0 | 25.9 | 25.9 | 1.00 | 1.04 | 0.92 | 0.80 |
| CG O3 | 25.9 | 25.9 | 1.00 | 1.05 | 0.94 | 0.79 |
| EP O0 | 0.02 | 0.02 | 1.00 | 1.00 | 1.00 | 1.00 |
| FT O0 | 12.8 | 12.8 | 1.00 | 1.04 | 0.97 | 0.94 |
| FT O1 | 12.9 | 12.8 | 1.00 | 1.02 | 0.95 | 0.92 |
| IS O0 | 28.4 | 42.4 | 0.67 | 1.01 | 1.01 | 1.67 |
| IS O1 | 36.5 | 36.3 | 1.00 | 1.02 | 0.96 | 0.94 |
| MG O0 | 62.9 | 43.3 | 1.45 | 1.06 | 0.99 | 0.93 |
| MG O3 | 63.6 | 44.9 | 1.42 | 1.06 | 0.92 | 0.82 |
| | | | Mean | 1.03 | 0.96 | 0.97 |

Table 1: Estimated L2 miss rates and speedups against default scheduling

overcomes the negative impact of bus contention between two threads. *Same-package* can never outperform *separate-package* since both have private caches but *Same-package* has bus contention. But *Same-package* can do better than *shared-L2* when *shared-L2* causes large cache conflicts.

Based on the heuristic equation developed in Section 4, we calculate a threshold of 0.79 based on the machine configuration we have. We estimate 9 front side bus cycles and an average of 18 memory cycles for each non-contention memory access. The bus contention adds 9 front side bus cycles. Based on the clock rate of the CPU core, FSB, and DRAM, we estimate 72 CPU cycles for a non-contention memory access and 88 cycles for a memory access with contention. The L2 hit latency is 13 cycles. Substituting the numbers into the equation produces 0.79. Only MM and IS O0 cross the threshold, which suggests *shared-L2* is the best choice for the two benchmarks. The results in Table 1 confirm this simple model. For MM and IS O0, *shared-L2* outperforms *separate-package* by 14% and 66% respectively. In the four-thread case, when every two threads are bound to a die (shared-L2) on separate processor, MM and IS O0 show 38% and 11% speedups compared with binding each thread to a separate die, while all other benchmarks show 0 to 12% slowdown. We want to point out the multi-core memory hierarchy is very complicated. Our heuristic model is still at its preliminary stage. However, the variation on execution times calls the attention of the compiler developers to explore the memory hierarchy and scheduling schemes further, which is one goal for us to develop this UPC compiler and share it with the research community.

## 5.4 Impact of "Out-of-Phase Execution"

Section 5.3 shows that binding two threads into one die may improve the performance when two threads have significant number of accesses to shared data. We observe that the L2 cache hit rates indeed vary across different runs under this scheme due to the *out-of-phase execution* described in Section 4. If the optimizing compiler can always obtain the high point of the variation, the execution can be stabilized at a higher speed. This section uses MM as an example to demonstrate this optimization. To verify our idea, we add an additional barrier by hand in the outermost loop of the 3-loop multiplication part in the original MM code to create a scenario that forces all threads to execute at a similar pace.

We compile both versions to two threads. We again use the simulator based on PIN to collect cache behavior of MM. We track cross-thread prefetches where one thread fetches shared data that are hit by the other thread. The prefetching rate is calculated as the ratio of the number of hits due to cross-thread prefetching to the total number of L2 accesses. Since the size of array $B$ is much larger than the L2 cache, the hits to L2 are mostly due to the prefetches.

Figure 5 shows the L2 hit rates and cross-thread prefetching rates in five runs. As can be seen, the "hit

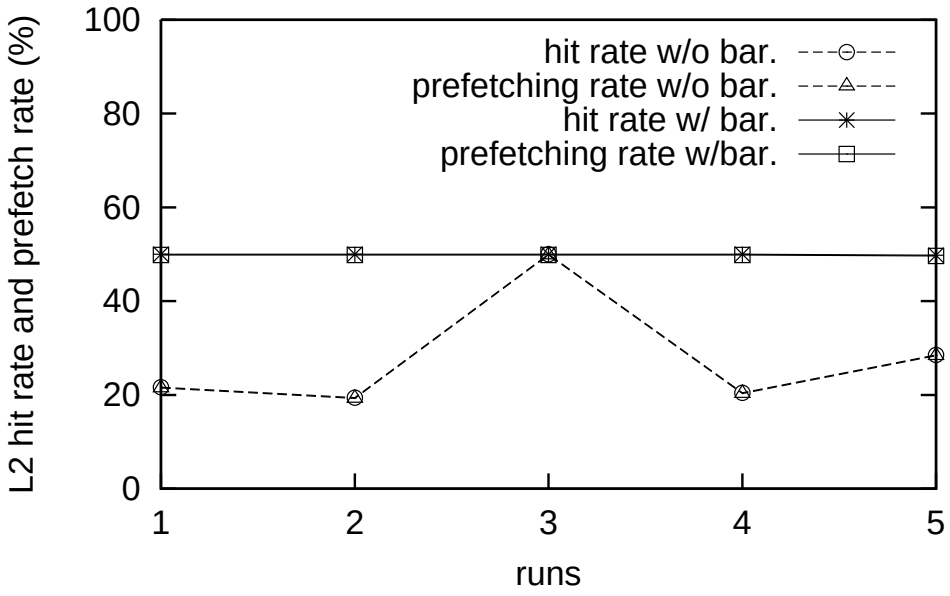



Figure 5: hit rates and prefetching rates of L2

rate" curves and the "prefetching rate" curves are perfectly overlapped for both versions, confirming that most L2 hits are from cross-thread prefetching. The hit rates with the extra barrier are steadily around 50%. On the contrary, without the extra barrier, the hit rates and prefetching rates vary significantly: they can be as low as 19% although they can reach 50% sometimes. When we directly run the two versions of MM five times using the *shared-L2* scheme, the average execution time for the version with the extra barrier is 6.73 seconds versus an average of 7.41 seconds for the original version, denoting a 9% improvement despite the cost of extra synchronization. If the compiler can predict this memory behavior, it can force synchronization to improve cache performance.

# 6 Conclusion and Future Work

The research community is actively seeking a well-accepted high level parallel programming language for multi-core computers. This paper adds our efforts by compiling UPC, a language designed for large scale parallel machines, for a multi-core architecture. We show that, under the block-major memory layout, our compiler can deliver comparable performance against the hand optimized code by minimizing the overhead of UPC pointer-to-shared arithmetic. We plan to develop a code generator in the middle end to take advantage of the existing analyses in the Scale compiler infrastructure. Our study shows that processor bindings can impact performance greatly and our profiling-based scheme can guide the bindings for optimal performance. Thread execution progress can also affect performance. Additional barriers to force threads synchronization can improve cross-thread prefetching rates and result in faster execution.

# References

[1] K. Yelick. (2004) Why UPC will take over OpenMP? [Online]. Available: http://www.openmp.org/drupal/node/view/13

[2] C. Barton, C. Carscaval, G. Almasi, Y. Zheng, M. Farrens, and J. Nelson, "Shared memory programming for large scale machines," in *Proceedings of the SIGPLAN 2006 Conference on Programming Language Design and Implementation*, Ottawa, Ontario, Canada, June 2006.

[3] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda, "An evaluation of global address space languages: co-array fortran and unified parallel c," in *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming.* New York, NY, USA: ACM Press, 2005, pp. 36–47.

[4] W.-Y. Chen, C. Iancu, and K. Yelick, "Communication optimizations for fine-grained upc applications," in *the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT '05)*, 17-21 Sept. 2005, pp. 267–278.

[5] T. El-Ghazawi and S. Chauvin, "UPC benchmarking issues," in *the 2001 International Conference on Parallel Processing (ICPP)*, 3-7 Sept. 2001, pp. 365–372.

[6] Berkeley UPC website. [Online]. Available: http://upc.lbl.gov

[7] (2004) The Intrepid UPC webiste. [Online]. Available: http://www.intrepid.com/upc

[8] Cray Inc. (2003) Cray X1 system overview. [Online]. Available: http://www.cray.com/craydoc/manuals/S-2346-22/html-S-2346-22/z1018480786.html

[9] The MTU UPC website. [Online]. Available: http://www.upc.mtu.edu

[10] Z. Zhang, J. Savant, and S. Seidel, "A UPC runtime system based on MPI and POSIX threads," in *14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing ( PDP '06)*, 15-17 Feb. 2006, p. 8pp.

[11] (2004) GASNet website. [Online]. Available: http://gasnet.cs.berkeley.edu

[12] W. Chen, D. Bonachea, J. Duell, P. Husbands, C. Iancu, and K. Yelick, "A performance analysis of the berkeley upc compiler," June 2003. [Online]. Available: citeseer.ist.psu.edu/article/chen03performance.html

[13] (2004) Compaq UPC for Tru64 UNIX. [Online]. Available: http://www.hp.com/go/upc

[14] M. S. Alexandra Fedorova and M. D. Smith, "Improving performance isolation on chip multiprocessors via an operating system scheduler," pp. 25–38, 2007.

[15] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture.* Washington, DC, USA: IEEE Computer Society, 2006, pp. 423–432.

[16] S. E. S. Parekh and H. Levy, "Thread-sensitive scheduling for smt processors," 2000. [Online]. Available: citeseer.ist.psu.edu/parekh00threadsensitive.html

[17] J. Nakajima and V. Pallipadi, "Enhancements for hyper-threading technology in the operating system: seeking the optimal scheduling," in *WIESS'02: Proceedings of the 2nd conference on Industrial Experiences with Systems Software.* Berkeley, CA, USA: USENIX Association, 2002, pp. 3–3.

[18] W.-T. L. Nauman Rafique and M. Thottethodi, "Architectural support for operating system-driven cmp cache management," in *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques.* New York, NY, USA: ACM, 2006, pp. 2–12.

[19] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 340–351.

[20] University of Massachusetts Amherst, "The Scale webiste." [Online]. Available: http://www-ali.cs.umass.edu/Scale/

[21] S. Siddha, V. Pallipadi, and A. Mallick, "Process scheduling challenges in the era of multi-core processors," *Intel Technology Journal*, vol. 11, pp. 361–369, 2007.

[22] UPC NAS benchmarks. [Online]. Available: http://www.gwu.edu/~upc/download.html

[23] (2007) Quad-core intel xeon processor 5300 series datasheet. [Online]. Available: http://www.intel.com/design/xeon/datashts/315569.htm

[24] (2007) Intel 5000 series chipset server board family datasheet. [Online]. Available: http://www.intel.com/support/motherboards/server/s5000psl/index.htm

[25] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance," Technical Report: NAS-99-011.

[26] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM Press, 2005, pp. 190–200.