

Runtime vs. Manual Data Distribution for Architecture-Agnostic Shared-Memory Programming Models

Dimitrios S. Nikolopoulos^{1*}, Eduard Ayguadé² and
Constantine D. Polychronopoulos¹

¹ Coordinated Science Laboratory
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
1308 W. Main St. Urbana, IL 61801
e-mail: {dsn,cdp}@csrd.uiuc.edu

² Department d' Arquitectura de Computadors
Universitat Politècnica de Catalunya
c/Jordi Girona 1-3, Modul D6
08034, Barcelona, Spain
e-mail: eduard@ac.upc.es

*Corresponding author, mailing address: 443-Coordinated Science Lab, University of Illinois at Urbana-Champaign, 1308 W. Main St., Urbana IL 61801. Phone: 217-244-5225, Fax: 217-244-1351

Abstract

This paper compares data distribution methodologies for scaling the performance of OpenMP on NUMA architectures. We investigate the performance of automatic page placement algorithms implemented in the operating system, runtime algorithms based on dynamic page migration, runtime algorithms based on loop scheduling transformations and manual data distribution. These techniques present the programmer with trade-offs between performance and programming effort. Automatic page placement algorithms are transparent to the programmer, but may compromise memory access locality. Dynamic page migration algorithms are also transparent, but require careful engineering and tuned implementations to be effective. Manual data distribution requires substantial programming effort and architecture-specific extensions to the API, but may localize memory accesses in a nearly optimal manner. Loop scheduling transformations may or may not require intervention from the programmer, but conform better to an architecture-agnostic programming paradigm like OpenMP.

We identify the conditions under which runtime data distribution algorithms can optimize memory access locality in OpenMP. We also present two novel runtime data distribution techniques, one based on memory access traces and another based on affinity scheduling of parallel loops. These techniques can be used to effectively replace manual data distribution in regular applications. The results provide a proof of concept that it is possible to scale a portable shared-memory programming model up to more than 100 processors, without modifying the API and without exposing architectural details to the programmer.

keywords: Data distribution, operating systems, runtime systems, performance evaluation, OpenMP.

1 Introduction

OpenMP [32] is a portable application programming interface (API) for developing multithreaded programs on any parallel architecture that provides the abstraction of a shared address space to the programmer. The target architecture may be a small-scale SMP server, a ccNUMA supercomputer, a cluster running software distributed shared-memory (DSM) [16, 23] or even a multithreaded processor [38]. The purpose of OpenMP is to ease the development of portable parallel code, by enabling incremental construction of parallel programs and shielding the programmer from the details of the underlying hardware/software interface. The philosophy of OpenMP is that the programmer should be able to obtain a scalable parallel program by adding parallelization directives to an already optimized sequential version of the program. Architectural features such as the communication substrate and the mechanism used to implement and orchestrate parallelism are hidden in the hardware and/or the runtime system and do not place any burden on the programmer. More importantly, the placement of data in memory is hidden from the programmer¹ and the assignment of computation to processors is done transparently via automatic worksharing constructs. OpenMP provides an *architecture-agnostic* API. Due to this property, the popularity of OpenMP has raised sharply [8] and OpenMP is now considered the *de facto* standard for programming shared-memory multiprocessors.

Unfortunately, the ease of programming with OpenMP is counterbalanced by the limited scalability on medium and large-scale multiprocessors with non-uniform memory access latency (NUMA). NUMA multiprocessors are built from basic components (called *nodes*), which include a handful of processors, memory and a communication assist. The physically distributed memory of the multiprocessor is shared among processors. However, memory is accessible with varying latencies, depending on the *distance* between processors and data in memory. In NUMA multipro-

¹The programmer must still consider the layout and the ordering of data in memory to exploit different forms of data locality and the caches, but the physical location of data is not exposed to the programmer in the parallelization process.

processors, local memory accesses cost several times less than remote memory accesses. The remote to local memory access latency ratio of state-of-the-art NUMA systems ranges between 3:1 and 10:1 [4, 11, 20, 22].

Since OpenMP provides no means to control the placement of data in memory, it is often the case that data is placed in memory in a manner that forces processors to issue a large number of remote memory accesses over the interconnection network. Localizing memory accesses via proper data distribution is a fundamental performance optimization for NUMA architectures and it is becoming more important as these architectures reach processor counts in the order of hundreds [13, 17]. Soon after realizing this problem, several researchers and vendors proposed and implemented data distribution extensions to OpenMP [2, 3, 10, 19, 21, 26, 35].

Integrating data distribution and preserving the architecture-agnostic programming interface are two conflicting requirements. Data distribution for NUMA architectures is essentially an architecture-dependent optimization. It requires significant programming effort, sometimes as much as the effort required to write a scalable message-passing program, in which data distribution is a mandatory part of the parallelization process. Data distribution for NUMA architectures requires also sophisticated compiler support [5]. Even simple regular distributions, such as block and cyclic, require several transformations including array padding, reshaping, and index rewriting to accurately place data in node memories. Due to these and other implications, data distribution is not currently integrated in the OpenMP API.

In recent work, we have shown that at least for iterative parallel programs with periodic memory access patterns (that is, programs that repeat the same parallel computation and access the same data for a number of iterations), the OpenMP runtime system can easily optimize data placement using a runtime data distribution engine [29]. The runtime data distribution algorithm that we proposed uses cost-effectiveness analysis to place each page with shared data in the node that minimizes the maximum latency of accesses to the page [28]. The cost-effectiveness analysis uses memory access traces that count the number of accesses from each processor to each page in

memory. Given that the memory access pattern is periodic and the program is iterative, a snapshot of memory accesses retrieved after the execution of one iteration of the program is sufficient to identify the best placement of pages. The actual page placement is done on-the-fly using dynamic page migration [39]. Alternatively, the algorithm runs off-line using the traces collected during the execution of the program. In this setting, the page placement produced by the algorithm is hardcoded in the program via recompilation.

After experimenting with our runtime data distribution engine, we established an argument that introducing data distribution extensions in OpenMP or other architecture-agnostic programming models might not be entirely necessary for scalable performance [28]. This argument has been validated with a synthetic experiment, in which OpenMP programs that were intentionally customized to the operating system’s page placement algorithm, were executed with alternative page placement algorithms. The experiment has shown that no matter what the initial distribution of data is, the runtime data distribution engine can accurately and timely relocate pages to match the performance of the best automatic page placement algorithm. This experiment indicated that shared-memory programming paradigms can possibly rely on runtime rather than manual data distribution to achieve the desired level of memory access locality on NUMA architectures. Given the implications of introducing data distribution to the OpenMP API and the emergence of OpenMP, we pursue this issue further, to investigate if runtime data distribution algorithms can indeed obviate the need for manual data distribution.

1.1 The Problems Addressed in this Paper

It is known that on NUMA architectures, manual data distribution and explicit assignment of computation to processors may improve considerably the performance of parallel programs [3, 5]. An important question that remains to be answered is whether runtime data distribution algorithms can provide the performance benefits of manual data distribution. Runtime data distribution algorithms

are sensitive to the granularity of the parallel program and potential irregularities in the memory access pattern. These factors may render runtime data distribution algorithms incapable of localizing memory accesses. In this paper we investigate runtime data distribution algorithms for regular applications. Irregular applications are treated elsewhere [31].

The algorithms presented in this paper target parallel programs that require regular data distributions to scale well on NUMA architectures. We borrow the term regular from HPF terminology [12], to characterize memory access patterns that can be localized using single-dimensional or multi-dimensional block and cyclic distributions. The objective is to avoid extending the OpenMP API with directives for data distribution and manual assignment of computation to processors. To this end, we present two runtime data distribution algorithms, one based on dynamic page migration and another based on affinity scheduling of parallel loops.

1.2 Summary of Contributions

This paper presents two runtime algorithms that implement regular data distributions in OpenMP programs either without programmer intervention. The first algorithm is based on our previous work on dynamic page migration [28, 29] and is used to implement single-dimensional block distributions, which can optimize memory access locality in many practical applications. The second algorithm is novel and uses affinity scheduling of parallel loops in conjunction with the first-touch page placement algorithm to implement arbitrary single-dimensional and multi-dimensional distributions.

We conducted an extensive set of experiments with the OpenMP implementations of the NAS benchmarks [1, 18] and a simple LU kernel on the NCSA Origin2000². We compared three versions of each benchmark: The unmodified OpenMP version, which has no *a priori* knowledge of the memory access pattern of the program and relies on the automatic page placement algorithm of the

²www.ncsa.uiuc.edu

operating system for data distribution; a hybrid data-parallel/OpenMP implementation that uses manual data distribution implemented with directives provided by the SGI FORTRAN compiler; and an OpenMP implementation which uses the proposed runtime data distribution algorithms.

The NAS benchmarks are popular codes used by the parallel processing community to evaluate scalable architectures. All NAS benchmarks need one-dimensional block distributions to localize memory accesses. The LU kernel requires cyclic distribution of the working array and cyclic loop scheduling for good data access locality and load balancing. We use it to evaluate the performance of the affinity scheduling algorithm.

The findings of the paper are summarized as follows. For regular, iterative parallel codes with periodic memory access patterns, the page migration algorithm matches or exceeds the performance of manual data distribution, if the granularity of the program is coarse enough to provide the page migration engine with a sufficiently long time frame for tuning data placement at runtime. In this case, dynamic page migration has two inherent advantages over manual data distribution, i.e. programming transparency and the ability to identify and place optimally falsely shared pages.

In fine-grain iterative codes with periodic memory access patterns, the page migration algorithm performs either in par or modestly worse than manual data distribution. It performs worse if the code has a certain degree of irregularity in the memory access pattern, which causes the page migration engine to produce an unbalanced placement of page across nodes. In these codes, using page migration at runtime or not is a matter of cost-benefit analysis, with the granularity of the parallel computation being the decisive factor. If page migration poses too much runtime overhead, the page placement algorithm can be applied at compile time. A memory access trace collected during a dry run of the program can be fed back to the compiler, which can run the data distribution algorithm offline and hardcode the placement of pages and any required array transformations in the optimized executable.

In the LU kernel, dynamic page migration harms performance. Fortunately, it is possible to achieve optimal memory access locality by scheduling the iterations of the innermost loop to pro-

processors in a cyclic manner and reusing this schedule in every iteration of the outermost sequential loop. This transformation implements a cyclic data distribution at runtime, by taking advantage of the first-touch page placement algorithm.

Overall, the results make a strong case for sophisticated runtime algorithms that implement transparently the performance optimizations required by architecture-agnostic programs. The paper contributes to a recently established belief that it is possible to use portable parallel programming paradigms and obtain scalable performance, thus simplifying the process and reducing the cost of developing efficient parallel programs. We consider our work as a step towards developing parallel programming models that yield the highest performance with the least programming effort.

1.3 The Rest of this Paper

The rest of this paper is organized as follows. Section 2 provides background and reviews related work. Section 3 presents our runtime data distribution algorithms. Section 4 presents our experimental setup and results. Section 5 summarizes our conclusions.

2 Background

The idea of introducing data distribution directives in shared-memory programming paradigms originates in the earlier work on data-parallel programming languages [12, 14]. Data-parallel languages use array distributions as the starting point of parallelization. Arrays are distributed along one or more dimensions, in order to let processors perform the bulk of the computation on local data and communicate rarely in a loosely synchronous manner. The communication is handled by the compiler, which has complete knowledge of the data owned by each processor. The computation is executed with the owner-computes rule, so that each processor computes using data available in local memory.

In order to use a data-parallel programming model on a NUMA multiprocessor, data distribution directives must be translated to distributions of the virtual memory pages that contain the array elements assigned to each processor. Communication is carried out directly through loads and stores in shared memory. From the performance perspective, the data-parallel programming style is attractive for NUMA architectures, because it establishes explicit bindings between computation and data. From the programming perspective though, data distribution compromises the simplicity of the shared-memory programming paradigm by exposing the data placement to the programmer. The implication is that the programming effort required to place data in memory and minimize the communication overhead might very well be as much as the effort required to obtain a scalable message-passing version of the program.

Several researchers and vendors have implemented data-parallel extensions to OpenMP [3, 5, 21, 26]. These extensions serve two purposes. The first is the distribution of data at page-level or element-level granularity. At page-level granularity, the unit of data distribution is a virtual memory page. At element-level granularity, the unit of data distribution is an individual array element. Since an element-level array distribution may assign two or more elements originally placed in the same page to different processors, the compiler may need to reshape arrays and rewrite subscripts to implement the distribution. The second purpose of data-parallel extensions to OpenMP is to explicitly assign loop iterations to specific memory modules to exploit locality. The objective is to assign each iteration to the memory module that contains all, or at least a good fraction of the data accessed by the loop body during that iteration. Data-parallel extensions of OpenMP have demonstrated significant performance improvements in simple numerical kernels like LU and SOR, albeit by modifying the native API and sacrificing portability [3, 5, 26].

We have developed a runtime system which applies dynamic locality optimizations to OpenMP programs [28, 30]. The runtime system records the number of accesses from each processor to each page in memory and applies competitive page migration algorithms at specific points of execution, where the access counters reflect accurately the memory access pattern of the program. In regular,

iterative parallel codes, these points are the ends of the outer iterations of the loop that encloses the parallel computation. This technique is both accurate and effective in relocating early at runtime the pages that concentrate many remote accesses. In most cases, the runtime system stabilizes the placement of pages after the execution of the first iteration and this placement is optimal with respect to the access pattern, in the sense that each page is placed in the node that minimizes the latency of accesses to the page [28]. The algorithm may actually perform better than manual data distribution, because it can remedy cases in which manual data distribution places pages in a non-optimal manner. This can happen for example if there are no appropriate distributions for certain memory access patterns that exist in the program, or if manual distribution places a falsely shared page with data assigned to multiple nodes in a node that does not access the pages more frequently than the other sharers.

The results obtained so far show that carefully engineered page migration algorithms can render OpenMP programs immune to the page placement algorithm of the operating system [29]. This means that programs perform always as good as they do with the best-performing automatic page placement algorithm, no matter how their data is initially placed in memory. Unfortunately, there are also cases in which manual data distribution outperforms the best automatic page placement algorithms by sizeable margins. It remains to be seen if a dynamic page migration engine can provide the same advantage over automatic page placement algorithms in such cases. A second important restriction of page migration is that it can only implement block data distributions and under certain constraints. Implementing arbitrary single-dimensional or multi-dimensional distributions at runtime requires additional support from the compiler and the runtime system.

3 Implementing Regular Data Distributions at Runtime

The algorithms presented in this section implement one-dimensional regular distributions (i.e. block and cyclic distributions) using a combination of runtime analysis of the memory access

pattern and loop transformations. The first algorithm implements block distributions in iterative codes with periodic access patterns, in which the work is distributed statically among processors. The second algorithm implements arbitrary regular distributions.

3.1 Block Data Distribution for Iterative Programs with Periodic Memory Access Patterns

In this section we show how our page migration engine can be used *as is* to implement single-dimensional block distributions for regular iterative codes. The page migration algorithm makes three assumptions about the structure of the program: First, it assumes that the program is *sequentially iterative* at the outer level. This means that the parallel computation executed by the program is enclosed in an outer sequential loop. Second, it assumes that the data access pattern of the program is *periodic*, in the sense that the program repeats the same sequence of data accesses across outer iterations. Third, it assumes that the parallel computation is *statically scheduled* and all loops in the program are equipartitioned among processors. The purpose of the algorithm is to implement a single-dimensional block distribution and in addition, place each page which is accessed from multiple processors in the node that minimizes the latency of memory accesses to the page.

A surprisingly large number of real parallel applications adhere to the three properties mentioned before. A representative example is the NAS benchmarks. The NAS benchmarks include computation kernels and simulation modules of real applications from the area of computational fluid dynamics. The codes are sequentially iterative and perform highly parallel computations that implement iterative methods for the numerical solution of differential equations. The methods execute for a number of time steps until the desired degree of convergence is achieved. The floating point FORTRAN codes from the SPEC CPU benchmark suite ^[37] meet also the aforementioned criteria. These codes cover a very broad spectrum of application domains, including weather pre-

diction, quantum physics, astrophysics, and hydrodynamic modelling, as well as more specialized numerical applications such as mesh generation, multigrid solvers and PDE solvers.

The algorithm works as follows: After the execution of one iteration of the computation, the runtime system collects histograms with the numbers of accesses from each node to each page in memory³. Using these histograms, the algorithm computes the maximum latency due to memory accesses to each page and the node to which the page should be moved, if necessary, to minimize this latency. The reader is referred to [28] for details on how the latency of memory accesses is accurately estimated, as well as an analysis of the migration criterion used by the algorithm.

This algorithm has three advantages. First, it is accurate, because the placement of pages is performed based on a complete snapshot of the memory access pattern of the program. Second, it amortizes well the cost of data distribution, because it is applied early during the execution of the program. For programs with several iterations, the algorithm can mask easily the overhead of moving data. Even if the cost of runtime data distribution is prohibitive, the algorithm is equally powerful, because the histograms can be collected during a probing run of the program and then provided as input to the compiler, for feedback-guided dynamic optimization. Third, the algorithm is transparent to the programming model and does not require any modifications to the API.

3.1.1 Implementation Issues

The major implementation issue of the algorithm is the counting of memory accesses per page. Memory accesses can be counted either in hardware or in software. Some commercial NUMA multiprocessors such as the SGI Origin series [20] and the Sun Wildfire [11] have vectors of hardware counters attached to each physical page in memory. Each of these vectors records the number of accesses from each node (or a small set of nodes) of the system to the page and is periodically flushed to buffers maintained by the operating system. The purpose of these counters is to help the

³We assume that accesses to a page are counted on a per-node rather than a per-processor basis, since a node may contain more than one processors and page placement is performed across nodes.

operating system implement its own dynamic page migration algorithm [39].

In systems with hardware page access counters, the algorithm can obtain memory access histograms for each page directly from the hardware. Our implementation of the algorithm on the Origin2000 uses the hardware counters, which are accessed via the `/proc` interface. This implementation takes advantage of the speed of counting in hardware to reduce drastically the cost of runtime data distribution, up to a point at which the algorithm can be applied while a program is running without performance loss. Our results in Section 4 show that this is indeed the case for long-running programs which can amortize the cost of dynamic data distribution across several iterations. In our implementation, the execution of the algorithm is overlapped with the execution of the program, using either a spare processor, or interleaving one processor between the data distribution thread and a thread of the program.

For systems without hardware counters, such as the Compaq GS/320 server [7], it is possible to instrument the program to count the accesses to elements of distributed arrays⁴. For each access to an array element, the instrumentation code records the physical memory address of the element and locates the page in which the element resides. The physical processor on which each thread is running is also recorded, for the proper association between processors and memory accesses. The page access counters are maintained in buffers allocated by the runtime system. Note that the technique of instrumenting memory accesses is not new and has been used before, e.g. to detect data dependencies in loops which are not analyzable for parallelization by a compiler [34]. Since tracing memory accesses in software is computationally expensive, this option can not be considered for runtime optimization. However, it is possible to use the algorithm for effective data distribution via dynamic compilation. The memory access trace is fed back to the compiler, which applies the algorithm, computes the best location for each page, and explicitly places the page to the specified location while generating code for the optimized version of the program. In the same

⁴We restrict ourselves to distributed arrays since we target numerical codes, in which most memory accesses are to array elements.

way, the algorithm can be used in codes with non-periodic memory access patterns, as long as the memory access pattern of the code remains the same across invocations and does not change with the input.

3.1.2 Using the Algorithm in Codes with Dynamic Scheduling

One important limitation of the algorithm is that it assumes that the parallel code is statically scheduled. This assumption ensures periodicity in the memory access pattern, either across iterations within the program, or across executions of the program on the same number of processors. Dynamic scheduling algorithms for parallel loops, such as guided self-scheduling [33], require synchronization to arbitrate the accesses of multiple processors to shared work queues. These queueing operations are required primarily for load balancing and render the schedule of work to processors non-deterministic.

A brute-force solution that deals with the problem of non-deterministic schedules is to record the schedule while the program is running. If a record of the schedule is available, the algorithm can proceed normally, by collecting the memory access trace, optimizing page placement and replaying the dynamic schedule in the optimized version of the program via static assignment of iterations to processors. The recording of the schedule amounts to updating an array of $P \times (N - P)$ bits, where P is the number of processors that execute the program and $N - P$ the number of iterations in the loop, assuming that each processor will execute at least one and at most $N - P$ iterations.

The problem with this solution is that the recording of the schedule is intrusive, since it affects the timing of synchronization operations and the schedule itself. It is possible to mask this effect to a certain extent, by performing the recording in the runtime system, while the runtime system dequeues work from the work queues and assigns iterations to processors. In this case, the recording overhead becomes part of the synchronization overhead. Assuming that the recording overhead is uniform across processors, chances are that recording will not affect the schedule significantly. This solution is still problematic though, because it requires access to the source code

of the OpenMP runtime system.

3.2 Implementing Generalized Regular Distributions with Affinity Scheduling

The second algorithm that we propose can be used to implement arbitrary regular distributions in OpenMP. The key idea of this algorithm is that it is possible to implement arbitrary distributions of data among processors by taking into account the algorithm used by the operating system to place pages in memory and customizing accordingly the loop scheduling algorithm. The algorithm exploits the first-touch page placement policy [6, 24]. First-touch places each page together with the processor that accesses the page first during the course of execution. The algorithm places a tile of data assigned to a specific processor by assigning the loop iterations that access the tile to the same processor, so that the processor touches the data first.

The algorithm makes three assumptions. First, data is distributed on a per-loop basis, i.e. data distribution is performed at loop-level granularity. Second, the distribution of computation in each loop matches the data distribution of exactly one of the arrays accessed in the loop. Let a be this array and assume that a is k -dimensional of size $n_1 \times n_2 \times \dots \times n_k$. The assumption implies that if an iteration of the loop accesses $a(i'_1 \dots i'_k)$, $1 \leq i'_1 \leq n_1, \dots, 1 \leq i'_k \leq n_k$, this iteration must be scheduled on the home node of $a(i'_1 \dots i'_k)$. Third, loops are parallelized along one-dimension, i.e. nested parallelism is not allowed.

Let us fix a parallel loop of nesting depth d , indexed with indices i_1, \dots, i_d and covering the iteration space $L_1 : U_{i_1} \dots L_d : U_d$, where L_i and U_i denote lower and upper bounds. Without loss of generality we assume that the loop is parallelized at the outermost level, i.e. the iterations $L_1 : U_1$ are distributed among processors. Suppose that a is accessed in the loop and data needs to be distributed somehow, so that while a processor executes its assigned part of the loop, it always finds the elements of a that it needs in local memory. The distribution of a can be modelled with

a mapping function $f : (i'_1, i'_2, \dots, i'_k) \rightarrow 1 \dots P$, where P is the number of processors. The function maps each element of the array to a processor and can be arbitrary, as soon as it ensures that each element is mapped to one processor. A cyclic distribution along the first dimension can be modelled as $f(i'_1, i'_2, \dots, i'_k) = i'_1 \bmod B$. Similarly, a block distribution along the first dimension can be modelled as $f(i'_1, i'_2, \dots, i'_k) = i'_1 / B$. Structured multidimensional and irregular distributions can be expressed in a similar fashion.

The algorithm starts with a data collection phase, during which it collects a trace with the elements of a accessed in every iteration of the parallel loop. The algorithm updates a k -dimensional array r of size equal to the size of a . Each entry $r(i'_1, \dots, i'_k)$ contains the index i_1 of the iteration that accesses $a(i'_1, \dots, i'_k)$, where $L_1 \leq i_1 \leq U_1$. Following the data collection phase, r contains a mapping $g : (i'_1, \dots, i'_k) \rightarrow L_1 : U_1$. The algorithm assigns to processor p all iterations i for which $r(i'_1, \dots, i'_k) = i \wedge f(i'_1, \dots, i'_k) = p$. This is done by constructing a dynamically allocated, asymmetric two-dimensional array r' , in which the first dimension is indexed with the processor number p and the second dimension contains as many entries as the number of iterations assigned to p . Assignment of iterations to processors is done at compile time, while generating a dynamically optimized version of the program.

One important optimization of the algorithm is that it memorizes loop schedules. If the same loop is executed more than once in the program, the algorithm reuses the schedule that implements the initial data distribution. In this way the algorithm exploits the affinity of processors to data, which is established during the first execution of the parallel loop [25]. To avoid introducing an extension to the OpenMP API, we use the algorithm to produce two versions of the program, one with block and one with cyclic distribution⁵. The compiler selects between the two distributions by timing the execution time of the two versions of the program.

⁵If multi-dimensional distribution is required, 2^d versions of the program are produced, where d is the number of distributed array dimensions.

Place Table 1 here.

3.2.1 Implementation Issues

The major implementation issue that has to be addressed in this algorithm is false sharing. Given that the unit of data distribution is a page, if the space occupied by each tile of the array assigned to a processor is not a multiple of the page size, one or more of the pages mapped to each processor will be falsely shared.

The problem of false sharing has no universal solution. For block and cyclic distributions, the problem can be solved with array reshaping and padding [5]. Reshaping amounts to adding a processor dimension to the array. Each processor accesses the data assigned to it by indexing the array with its virtual *id*. Padding is simpler and amounts to adding a number of zero-filled elements to certain dimensions of the array, so that the tiles of the array assigned to each processor are page-aligned in memory. In our implementation we use padding for the sake of efficiency, since reshaping involves potentially expensive division and modulo operations to compute the reshaped array indices. Although the transformation is applied by hand for the purposes of experimentation, there exists mature compiler technology that would easily automate this procedure.

4 Experimental Setup and Results

We conducted two sets of experiments, one using the NAS benchmarks and one using a simple hand-crafted LU kernel. The experiments were executed on a 128-processor cluster of the NCSA Origin2000. This cluster uses MIPS R10000 processors clocked at 250 MHz, with 32 Kbytes of split L1 cache and 4 Mbytes of unified L2 cache per processor. The cluster has 64 Gbytes of uniformly distributed DRAM memory. The experiments were conducted on dedicated processors.

4.1 NAS benchmarks

In the first set of experiments we executed five benchmarks from the NAS suite, implemented in OpenMP by researchers at NASA Ames [18]. We used the class A problem sizes, which fit the scale of the system on which we ran the experiments. The benchmarks are BT, SP, CG, FT and MG. BT and SP are complete CFD simulations, while CG, MG and FT are computational kernels extracted from real CFD codes. BT and SP solve uncoupled Navier-Stokes equations in three dimensions, using the Alternate Direction Implicit (ADI) method and the Beam Warming approximate factorization method respectively. FT is the computational kernel of a three-dimensional FFT-based spectral method. MG uses a multigrid method to solve a three-dimensional, scalar Poisson equation. CG approximates the smallest eigenvalue of a sparse matrix using the Conjugate-Gradient method. We used the Class A problem sizes, which are shown in Table I. These problem sizes are large enough for the scale of the system on which we experimented.

The OpenMP implementations of the benchmarks are tuned by their providers specifically for the Origin2000 memory hierarchy [18]. The codes include a cold-start iteration of the parallel computation that distributes data among processors on a first-touch basis. The cold-start iteration performs a block distribution of the dimension of the arrays accessed in the outermost level of parallel loop nests. Its functionality is equivalent to that of a *DISTRIBUTE (*,...*,BLOCK)* directive in HPF. The SGI Fortran compiler provides data distribution directives that implement single-dimensional and multi-dimensional regular distributions. The directives resemble those of HPF and their implementation is described in [5]. In addition to these directives, the compiler provides an optional affinity clause, which can be used to specify the processor on which each iteration of a parallel loop should be executed.

Since data distribution was already hard-coded in the benchmarks, in order to evaluate the performance of the same benchmarks without manual data distribution we use the following setting. We remove the cold-start iteration and use the default page placement algorithm of our hardware

platform, which in the case of the NCSA Origin2000 is round-robin. In this situation, we artificially setup a scenario in which the automatic page placement algorithm of the operating system does not distribute data properly. The reason for devising such a scenario is that for many applications and systems, the default data placement algorithms provided by the operating system do not match the data access pattern of the programs. The experiment captures these cases and demonstrates what may happen if the programmer is not —or does not want to be— aware of an architecture-specific feature, such as the non-uniform memory access latency, while developing the program. Running the experiment is as simple as setting an environment variable (*_DSM_PLACEMENT*) to the value *ROUND_ROBIN* and executing the code as is. Note that in earlier work [28] we have shown that round-robin is not a bad choice for a default data placement algorithm. In fact, it has quite good performance in many regular applications, primarily because it balances well the memory accesses across the nodes of the system.

We executed three versions of each benchmark:

- *OpenMP_automatic*: This is the original OpenMP code without the cold-start iteration, executed with round-robin page placement. This version corresponds to a case in which the programmer has absolutely no knowledge of the placement of data in memory and the OS uses a non-optimal automatic page placement algorithm. The performance of this version serves as a baseline for comparisons.
- *OpenMP_manual*: This is the original OpenMP code modified with *!\$SGI DISTRIBUTE* directives to perform manual data distribution of shared arrays. The pages not included in distributed arrays are distributed by the operating system on a round-robin basis.
- *OpenMP_runtime*: This is the native OpenMP code, linked with our runtime system. We classify these implementations further into implementations that use dynamic page migration (*OpenMP_runtime_online*) and implementations that hardcode the optimized data placement after collecting a trace of memory accesses and recompiling (*OpenMP_runtime_offline*). For

Place Table 2 here.

Place Figure 1 here.

the time being, we have implemented this setting by hand using a custom code transformation tool.

The implementation of the *OpenMP_manual* version follows the guidelines from the HPF implementation of the NAS benchmarks, originally described in [9]. The distributions were implemented with the SGI compiler’s *DISTRIBUTE* directive [5], the semantics of which are similar to the corresponding HPF directive. We emphasize that in this implementation, the data distribution overhead is not on the critical path, therefore it is not reflected in the measurements.

Table II shows the data distributions applied to the benchmarks. Two benchmarks, BT and SP, might benefit from using data redistribution within iterations. The memory access patterns of these benchmarks have phase changes. In both benchmarks, there is a phase change between the execution of the solver in the *y*-direction and the execution of the solver in the *z*-direction, due to the initial placement of data, which exploits spatial locality along the *x*- and *y*- directions. We implemented versions of BT and SP in which the arrays were *BLOCK*-redistributed along the *y*-dimension at the entry of subroutine *z_solve* and *BLOCK*-redistributed again along the *z*-direction at the exit of *z_solve*. These redistributions were implemented using the *REDISTRIBUTE* directive. Although theoretically beneficial for BT and SP, the redistributions increased the execution time of the benchmarks to unacceptable levels⁶. The results reported in Section 4 are obtained from experiments without data redistribution.

The results from the experiments with the NAS benchmarks are summarized in Figures 1 and 2. Figure 1 shows the execution times of the benchmarks on a varying number of processors. The *y*-axis in the charts is drawn in logarithmic scale, the *x*-axis is not linear, and the maximum values of the *y*-axis are adjusted according to the sequential execution times of the benchmarks for the

⁶The overhead of data redistribution may be attributed to a poor implementation [20].

Place Figure 2 here.

sake of readability. Note that in some cases superlinear speedups occur due to cache effects. More specifically, superlinear speedups occur in CG on 8 and 16 processors, and in FT on 8, 16, and 32 processors.

Figure 2 shows histograms of memory accesses per-node, divided into local and remote memory accesses⁷ for two benchmarks, BT and MG. These benchmarks illustrate the most representative trends in the experiments. The histograms are obtained from instrumented executions of the programs on 128 processors (64 nodes of the Origin2000). The instrumentation code calculates the total number of local and remote memory accesses per-node, by reading the page access counters in the entire address space of the programs.

The results from the experiments with the NAS benchmarks show a consistent trend of improvement when manual data distribution is added to the OpenMP implementation. The trend is observed clearly from the 16-processor scale and beyond. At this scale, the ratio of remote-to-local memory access latency approaches 2:1 and data placement becomes critical. On 16 processors, data distribution reduces execution time by 5.4% on average. On 128 processors, data distribution improves performance by a wider margin, which averages 25% and in three cases (MG, CG, and FT) exceeds 30%.

The difference between manual data distribution and round-robin page placement is purely a matter of memory access locality. Figure 2 shows that in BT, round-robin placement of pages forces 55% of memory accesses to be remote. The *BLOCK* distribution reduces the fraction of remote memory accesses to 25%. Notice that this fraction is still significant enough to justify further optimization. In BT, it amounts to approximately one million remote memory accesses per processor. The savings from reducing remote memory accesses are significant, not only because the latency of memory accesses is reduced, but also because the localization of memory accesses

⁷Remote in the sense that the node is accessed remotely by processors in other nodes.

reduces contention at the memory modules and network interfaces. Contention accounts for an additional overhead of approximately 50 ns per contending node per access on the Origin2000 [15] and it has been shown that it can have a very significant impact on performance [27].

The behavior of the runtime data distribution algorithm is explained by classifying the benchmarks in two classes, coarse-grain and fine-grain benchmarks. In this context, granularity is associated with the duration of the parallel computation and the number of iterations that the program executes.

4.1.1 Coarse-grain Benchmarks with Regular Memory Access Patterns

The first class includes benchmarks with a large number of iterations, relatively long execution times (in the order of several hundreds of milliseconds per iteration) and regular memory access patterns. The term *regular* refers to the distribution of memory accesses, i.e. the memory accesses, both local and remote, are uniformly distributed across nodes. BT and SP belong to this class. Figure 2 illustrates the regularity in the access pattern of BT. SP has a very similar access pattern.

In these benchmarks, the page migration engine matches the performance of manual data distribution. The memory access histograms in Figure 2 show that the runtime data distribution algorithm actually reduces the number of remote memory accesses by a small margin. This happens because the algorithm reduces the number of remote memory access to falsely shared pages, which occur because the data tiles assigned to each processor are not page-aligned.

Despite the reduction of remote memory accesses when data distribution is implemented with page migration, there is no significant advantage over manual data distribution. This happens because the gain from reducing memory latency is offset by the overhead of page migration. Applying the algorithm in a dynamic compilation setting overcomes this limitation by removing the overhead of data distribution from the critical path. This yields a measurable improvement over manual data distribution on a large number of processors.

4.1.2 Fine-grain Benchmarks with Potentially Irregular Memory Access Patterns

The second class includes fine-grain benchmarks with a small number of iterations. CG, MG and FT belong to this class. Runtime data distribution with dynamic page migration performs in par or outperforms manual data distribution only in CG. In FT and MG, manual data distribution performs slightly better than runtime data distribution. This is attributed to the granularity of the computation. If the execution time per-iteration is too short and the benchmark executes only a handful of iterations, the page migration engine does not have enough time to migrate badly placed pages in a timely manner, if at all. Note that CG executes 15 iterations, while MG and FT execute only 4 and 6 iterations respectively.

MG exposes the problem more clearly. MG executes only 4 iterations, the runtime on 128 processors is no more than a second and as shown in Figure 2, the benchmark has an irregularity in the memory access pattern, in the sense that memory accesses are not distributed evenly across nodes. Manual data distribution alleviates this irregularity and produces a balanced distribution of both local and remote memory accesses. Runtime data distribution with dynamic page migration reduces the number of remote memory accesses by 36%, but does not alleviate the irregularity of the memory access pattern. The pattern remains visually unaltered, albeit with fewer remote accesses. The accumulated number remote memory accesses in the *OpenMP_manual* version and the *OpenMP_runtime* is approximately the same, however, in the *OpenMP_manual* version, the remote memory accesses are evenly balanced across processors, whereas in the *OpenMP_runtime* version there is a noticeable degree of memory load imbalance. Careful examination of the traces shows that in the *OpenMP_runtime* version there is an unbalance in the number of pages placed in each node, which in turn causes some nodes to concentrate more contested pages⁸ than other nodes. We believe that this problem can be solved by careful engineering of the page migration criterion in our algorithms, however this is outside the scope of this paper.

⁸Contested in the sense that the pages have many remote accesses.

Place Figure 3 here.

Place Figure 4 here.

Another problem with MG is the overhead of runtime data distribution. Figure 3 shows the overhead of the page migration engine, normalized to the execution times of the benchmarks on 128 processors. This overhead is also computed with instrumentation. It must be noted that the runtime system overlaps the execution of page migration with the execution of the program. A thread which is activated periodically is used to run the page migration algorithm in parallel with the program. Although the overhead of page migration is masked, the thread used by the runtime system does interfere with the threads of the OpenMP program and this interference becomes noticeable when the programs use all 128 processors of the system. We conservatively assume that this interference causes a 50–50 processor sharing and we estimate the overhead of the runtime system as 50% of the total CPU time consumed by the thread that executes the page migrations. The chart shows that on 128 processors, the interference between the thread that runs the page migration algorithm and the OpenMP threads accounts for 39% of execution time in MG. A closer look at the page migration activity of MG revealed that the runtime system executes almost 1400 page migrations. The runtime system has noticeable overhead also in FT (20% of the total execution time).

To investigate why the page migration engine is inferior to manual data distribution in MG and FT, we conducted a synthetic experiment, in which we scaled the benchmark by doubling the number of iterations. This modification does not affect the memory access pattern or the problem size. Its purpose is to check if the cost of runtime data distribution can be amortized by executing more iterations. The result of this experiment is shown in Figure 4.

Figure 4 shows two different trends. In MG, runtime data distribution seems unable to match the performance of manual data distribution, despite the coarser granularity. At a first glance, this result seems surprising. Timing instrumentation of the runtime system indicates that in the scaled

Place Figure 5 here.

Place Figure 6 here.

version of MG, the overhead of page migration accounts only for 23% of the total execution time, which is a little less than half of the overhead observed in the non-scaled version. Unfortunately, as Figure 5 shows, the fraction of remote memory accesses of MG does not seem to decrease with runtime data distribution, despite the coarser granularity of the program. FT presents a different picture. Although there is some memory load imbalance, increasing the number of iterations enables stability of page placement by the migration engine in the long-term. The relative overhead of page migration in the scaled version of FT does not exceed 7% of the total execution time.

In all three fine-grain benchmarks (CG, MG and FT), using the data distribution algorithm at compile time after collecting the trace of memory accesses yields performance which matches or exceeds the performance of manual data distribution (as shown by the execution times of the *OpenMP_runtime_offline* version in Figure 1). Therefore, it is not necessary to use manual data distribution. This verifies the effectiveness of the algorithm and shows that it can serve as a powerful tool for dynamic optimization of unmodified OpenMP code.

4.2 LU

The LU kernel shown in Figure 6 (left fragment) is hard to optimize with dynamic page migration. Although the code is iterative, the amount of computation performed in each iteration is progressively reduced and the data touched by a processor may differ from iteration to iteration, depending on the algorithm that assigns iterations to processors. A block distribution can not improve memory access locality significantly. Cyclic distribution along the second dimension of a is more helpful because it distributes evenly the computation across processors. In order to have both balanced load and good memory access locality though, the code should be modified as shown in the right fragment in Figure 6. In addition to the cyclic distribution, the loop should be scheduled

Place Figure 7 here.

Place Figure 8 here.

so that each iteration of the j loop is executed on the node where the j -th column of a is stored. This is accomplished with the *AFFINITY* clause of the SGI compiler, which is the analogue of the *ONHOME* clause in HPF. Note that the loop schedule is maintained across iterations of the k loop.

We have implemented four versions of LU. Three of these versions, *OpenMP_automatic*, *OpenMP_manual* and *OpenMP_runtime* are produced in the same manner as the implementations of the NAS benchmarks described in Section 4.1. In the *OpenMP_runtime_online* version, page migration is applied at the end of iterations of the k loop. The fourth version (*OpenMP_runtime_reuse* shown in Figure 7) uses our affinity scheduling transformation to implement runtime data distribution and proper assignment of work to processors. In this version, the j loop is transformed into a loop iterating from 0 to $nprocs-1$, where $nprocs$ is the number of processors executing in parallel. Each processor computes independently the set of iterations to execute. Iterations are assigned to processors in a cyclic manner and during the k -th iteration of the outer loop, a processor executes a subset of the iterations that the same processor executed during the $(k-1)$ -th iteration of the outer loop.

The purpose of the cyclic assignment of iterations is to have each processor reuse the data that it touches during the first iteration of the k loop. If the program is executed with first-touch page placement, such a transformation achieves good localization of memory accesses. In some sense, the transformation resembles the affinity loop scheduling algorithm proposed in [25]. The difference is that the transformation is used to exploit cache reuse and at the same time, localize memory accesses. The LU kernel was executed on a dense 1400×1400 matrix.

Figure 8 illustrates the execution times of different versions of LU. The chart shows that runtime data distribution across iterations of the outer loop of LU does not provide any improve-

Place Figure 9 here.

ment over round-robin page placement. The version that uses cyclic data distribution and affinity scheduling (*OpenMP_manual*) outperforms both the version that uses round-robin page placement (*OpenMP_automatic*) and the version that uses dynamic page migration (*OpenMP_runtime_online*). This indicates that load balancing is important and that any localization of memory accesses by the page migration engine is either ineffective, or imposes overhead which overwhelms the benefit of reduced memory access latency. Timeliness is the major problem of the page migration algorithm in LU. Although the page migration engine is able to detect the iterative shifts of the memory access pattern, it is unable to migrate pages ahead in time to reduce the impact of these shifts. The *OpenMP_runtime_reuse* version matches or exceeds slightly the performance of the manually optimized version.

Figure 9 shows the memory access traces of three versions of LU, the *automatic*, the *manual* and the *reuse* version. The histograms show that the loop transformation is as effective as manual data distribution, as far as the number of remote memory accesses is concerned. This happens because each processor computes repeatedly on data that the processor touches first during the execution of the program. The transformation exploits also cache block reuse.

5 Conclusions

We conducted a detailed evaluation of several data distribution methods for improving the scalability of OpenMP on NUMA multiprocessors. The simplest solution is to use the existing automatic page placement algorithms of the operating system, which include some NUMA-aware algorithms such as round-robin and first-touch. The experiments have shown that this solution is not adequate. At the moderate scale of 16 processors, automatic page placement algorithms already perform worse than manual data distribution, whereas at larger scales, the performance

of automatic page placement algorithms shows diminishing returns. The second step to attack the problem, i.e. introducing data distribution directives in OpenMP, reaches performance levels which verify the common belief that a data-parallel programming style is likely to solve the problem of memory access locality on NUMA architectures. Nevertheless, the experiments show that manual data distribution is not a panacea. The same or better performance can be obtained from carefully engineered runtime data distribution algorithms, based on dynamic page migration and affinity scheduling.

Dynamic page migration is effective under certain constraints regarding the granularity, regularity and periodicity of the memory access pattern. These constraints are important enough to justify further investigation, since several parallel codes do not possess all these properties. Our experiments with LU, a program with an aperiodic but fairly simple access pattern, gave us a hint that loop schedules with affinity links maintained across invocations of the same loop are able to localize memory accesses very well. We believe that several codes might benefit from flexible affinity scheduling strategies that move computation close to data, rather than data close to computation. It is a matter of further investigation and experimentation to verify this assumption.

The problem of granularity is hard to overcome because the overhead of page migration remains high⁹. It may be worth the effort to investigate alternatives such as virtually addressed cache organizations that move the TLB out of the processor core and down in the memory hierarchy. Such organizations can greatly reduce the TLB consistency overhead, which dominates the page migration time. A software alternative is to parallelize the page migration procedure. The idea is to have each processor poll the access counters of locally mapped pages and forward pages that satisfy the migration criterion to the receiver. We expect that careful inlining of the page migration algorithm in OpenMP threads will reduce the interference between the page migration engine and the program and hopefully provide more opportunities for page migration under tight time constraints. At the very end, this paper proposed an alternative methodology based on dynamic

⁹The cost for migrating a page on the Origin2000 is approximately 1 ms.

compilation, which removes the overhead of runtime data distribution out of the critical path.

Irregular codes are a challenging domain for future work. Adaptive codes for example, have memory access patterns which are a-priori unknown and change at runtime in an unpredictable manner [36]. Irregular parallel applications appear to be one class of programs where page migration is not an option and domain-specific knowledge may be required to encode proper algorithms for data and load distribution.

Acknowledgements

This work was supported in part by the E.C. TMR Contract No. ERBFMGECT-950062, NSF Grant No. EIA-9975019 and the Spanish Ministry of Education Grant No. TIC98-511. The experiments were conducted with resources provided by NCSA at the University of Illinois, Urbana-Champaign.

References

- [1] D. Bailey, T. Harris, W. Saphir, R. V. der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-020, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Dec. 1995.
- [2] S. Benkner and T. Brandes. Exploiting Data Locality on Scalable Shared Memory Machines with Data Parallel Programs. In *Proc. of the 6th International EuroPar Conference (EuroPar'2000)*, pages 647–657, Munich, Germany, Aug. 2000.
- [3] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. Nelson, and C. Offner. Extending OpenMP for NUMA Machines. In *Proc. of the IEEE/ACM Supercomputing'2000: High Performance Networking and Computing Conference (SC'2000)*, Dallas, Texas, Nov. 2000.
- [4] T. Brewer and J. Astfalk. The Evolution of the HP/Convex Exemplar. In *Proc. of the 42nd IEEE Computer Society International Conference (COMPCON'97)*, pages 81–96, San Jose, California, Feb. 1997.
- [5] R. Chandra, D. Chen, R. Cox, D. Maydan, N. Nedeljkovic, and J. Anderson. Data Distribution Support on Distributed Shared Memory Multiprocessors. In *Proc. of the 1997 ACM Conference on Programming Languages Design and Implementation (PLDI'97)*, pages 334–345, Las Vegas, Nevada, June 1997.

- [6] R. Chandra, S. Devine, A. Gupta, and M. Rosenblum. Scheduling and Page Migration for Multiprocessor Compute Servers. In *Proc. of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 12–24, San Jose, California, Oct. 1994.
- [7] Compaq Computer Corporation. Compaq Alpha Server GS/320 System Technical Summary. <http://www.compaq.com/alphaserver>, May 2000.
- [8] cOMPunity. The Community of OpenMP Users, Researchers, Tool Developers, and Providers. <http://www.compunjity.org>, 2002.
- [9] M. Frumkin, H. Jin, and J. Yan. Implementation of NAS Parallel Benchmarks in High Performance FORTRAN. Technical Report NAS-98-009, NASA Ames Research Center, Sept. 1998.
- [10] W. Gropp. A User’s View of OpenMP: The Good, The Bad and the Ugly. In *Workshop on OpenMP Applications and Tools (WOMPAT’2000)*, San Diego, California, July 2000.
- [11] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proc. of the 5th International Symposium on High Performance Computer Architecture (HPCA-5)*, pages 171–181, Orlando, Florida, Jan. 1999.
- [12] High Performance FORTRAN Forum. High Performance FORTRAN Language Specification, Version 2.0. Technical Report CRPCTR-92225, Center for Research on Parallel Computation, Rice University, Jan. 1997.
- [13] C. Holt, J. P. Singh, and J. Hennessy. Application and Architectural Bottlenecks in Large-Scale Distributed Shared Memory Machines. In *Proc. of the 23rd International Symposium on Computer Architecture (ISCA’96)*, pages 134–145, Philadelphia, Pennsylvania, June 1996.
- [14] HPF+ Project Consortium. HPF+: Optimizing HPF for Advanced Applications. <http://www.par.univie.ac.at/project/hpf+>, 1998.
- [15] C. Hristea, D. Lenoski, and J. Keen. Measuring Memory Hierarchy Performance on Cache-Coherent Multiprocessors Using Microbenchmarks. In *Proc. of the ACM/IEEE Supercomputing’97: High Performance Networking and Computing Conference (SC’97)*, San Jose, California, Nov. 1997.
- [16] Y. Hu, H. Lu, A. Cox, and W. Zwaenepoel. OpenMP on Networks of SMPs. In *Proc. of the 13th International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP’99)*, pages 302–310, San Juan, Puerto Rico, Apr. 1999.
- [17] D. Jiang and J. P. Singh. Scaling Application Performance on a Cache-Coherent Multiprocessor. In *Proc. of the 26th International Symposium on Computer Architecture (ISCA’99)*, pages 305–316, Atlanta, Georgia, May 1999.

- [18] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of the NAS Parallel Benchmarks and its Performance. Technical Report NAS-99-011, NASA Ames Research Center, Oct. 1999.
- [19] D. Kuck. OpenMP: Past and Future. In *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT'2000)*, San Diego, California, July 2000.
- [20] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA'97)*, pages 241–251, Denver, Colorado, June 1997.
- [21] J. Levesque. The Future of OpenMP on IBM SMP Systems. In *Proc. of the First European Workshop on OpenMP (EWOMP'99)*, pages 5–6, Lund, Sweden, Oct. 1999.
- [22] T. Lovett and R. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proc. of the 23rd International Symposium on Computer Architecture (ISCA'96)*, pages 308–317, Philadelphia, Pennsylvania, May 1996.
- [23] H. Lu, Y. Hu, and W. Zwaenepoel. OpenMP on Networks of Workstations. In *Proc. of the IEEE/ACM Supercomputing'98: High Performance Networking and Computing Conference (SC'98)*, Orlando, Florida, Nov. 1998.
- [24] M. Marchetti, L. Kontothanassis, R. Bianchini, and M. Scott. Using Simple Page Placement Schemes to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems. In *Proc. of the 9th IEEE International Parallel Processing Symposium (IPPS'95)*, pages 380–385, Santa Barbara, California, Apr. 1995.
- [25] E. Markatos and T. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, Apr. 1994.
- [26] J. Merlin and V. Schuster. HPF-OpenMP for SMP Clusters. In *Proc. of the 4th Annual HPF User Group Meeting (HPFUG'2000)*, Tokyo, Japan, Oct. 2000.
- [27] D. Nikolopoulos. Quantifying and Resolving Remote Memory Access Contention on Hardware DSM Multiprocessors. In *Proc. of the 16th IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS'02)*, Fort Lauderdale, Florida, Apr. 2002.
- [28] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta, and E. Ayguadé. A Case for User-Level Dynamic Page Migration. In *Proc. of the 14th ACM International Conference on Supercomputing (ICS'2000)*, pages 119–130, Santa Fe, New Mexico, May 2000.
- [29] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta, and E. Ayguadé. Is Data Distribution Necessary in OpenMP ? In *Proc. of the IEEE/ACM Supercomputing'2000: High Performance Networking and Computing Conference (SC'2000)*, Dallas, Texas, Nov. 2000.

- [30] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta, and E. Ayguadé. UPM-lib: A Runtime System for Tuning the Memory Performance of OpenMP Programs on Scalable Shared-Memory Multiprocessors. In *Proc. of the 5th ACM Workshop on Languages, Compilers and Runtime Systems for Scalable Computers (LCR'2000)*, LNCS Vol. 1915, pages 85–99, Rochester, New York, May 2000.
- [31] D. Nikolopoulos, C. Polychronopoulos, and E. Ayguadé. Scaling Irregular Parallel Codes with Minimal Programming Effort. In *Proc. of ACM/IEEE Supercomputing'2001: High Performance Networking and Computing Conference (SC'2001)*, Denver, Colorado, Nov. 2001.
- [32] OpenMP Architecture Review Board. OpenMP Fortran Application Programming Interface. Version 1.2, <http://www.openmp.org>, Nov. 2000.
- [33] C. Polychronopoulos and D. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, C-36(12):1485–1495, Dec. 1987.
- [34] L. Rauchwerger and D. Padua. The Privatizing DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization. In *Proc. of the 8th ACM International Conference on Supercomputing (ICS'94)*, pages 33–43, Manchester, United Kingdom, July 1994.
- [35] V. Schuster and D. Miles. Distributed OpenMP, Extensions to OpenMP for SMP Clusters. In *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT'2000)*, San Diego, California, July 2000.
- [36] H. Shan, J. P. Singh, R. Biswas, and L. Oliner. A Comparison of Three Programming Models for Adaptive Applications on the Origin2000. In *Proc. of the IEEE/ACM Supercomputing'2000: High Performance Networking and Computing Conference (SC'2000)*, Dallas, Texas, Nov. 2000.
- [37] Standard Performance Evaluation Corporation. SPEC CPU2000 Benchmarks. <http://www.spec.org>, Dec. 2000.
- [38] X. Tian, A. Bik, M. Girkar, P. Grey, H. Saito, and E. Su. Intel OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance. *Intel Technology Journal*, 6(1), Jan. 2002.
- [39] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 279–289, Cambridge, Massachusetts, Oct. 1996.

NAS BT	$64 \times 64 \times 64$ grid
NAS SP	$64 \times 64 \times 64$ grid
NAS FT	$256 \times 256 \times 128$ grid
NAS MG	$256 \times 256 \times 256$ grid
NAS CG	14000-non-zero element array
IFS LG	$63 \times 63 \times 63$ grid
IFS SL	$63 \times 63 \times 63$ grid
IFS TS	$63 \times 63 \times 63$ grid

Table I: Benchmarks and problem sizes used.

Benchmark	Distributions
BT	<i>BLOCK</i> , last dimension
SP	<i>BLOCK</i> , last dimension
CG	one-dimensional <i>BLOCK</i>
FT	one-dimensional <i>BLOCK</i>
MG	one-dimensional <i>BLOCK</i>

Table II: Data distributions applied to the OpenMP implementations of the NAS benchmarks.

Figure 1: Execution times of the NAS benchmarks.

Figure 2: Memory access histograms of BT and MG. The memory accesses are divided into local (gray) and remote (black) accesses.

Figure 3: Relative overhead of the page migration engine in the NAS benchmarks (black part). The overhead is normalized to the execution time of each benchmark.

Figure 4: Execution times of the scaled versions of MG and FT.

Figure 5: Memory access histograms of the scaled version of MG.

Figure 6: LU kernel implemented with standard OpenMP (left) and with data distribution and affinity scheduling (right).

Figure 7: LU with iteration schedule reuse.

Figure 8: Execution times of LU.

Figure 9: Memory access histograms of LU. The memory accesses are divided into local (gray) and remote(black) accesses.

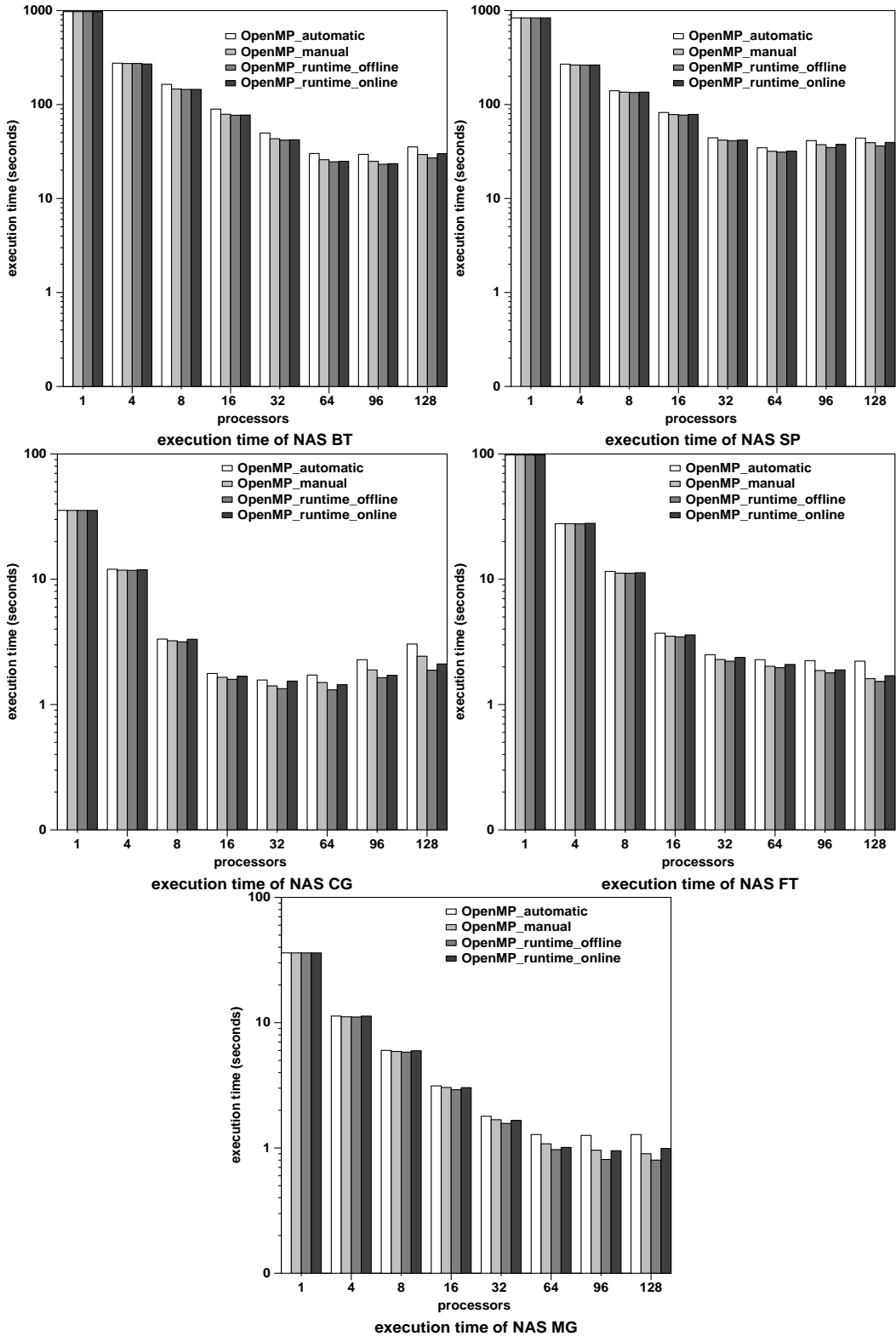


Figure 1

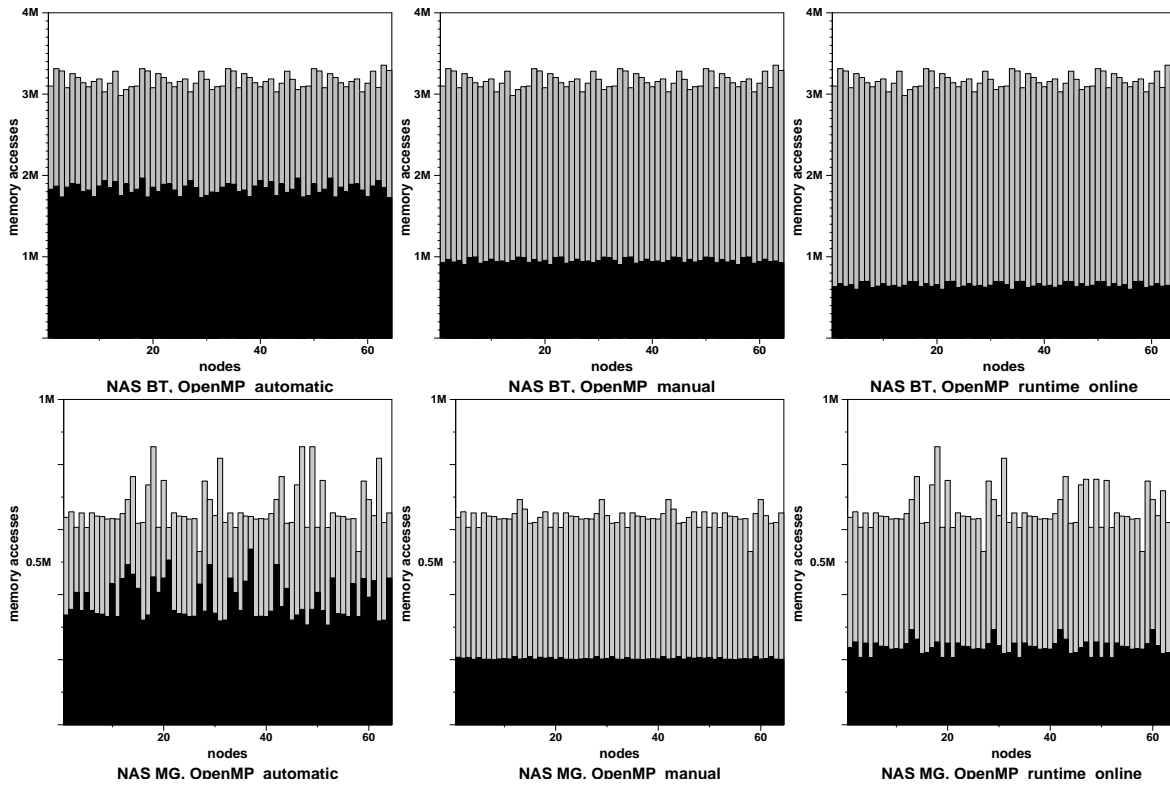


Figure 2

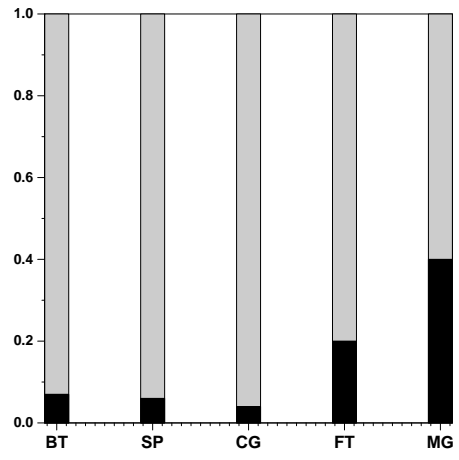


Figure 3

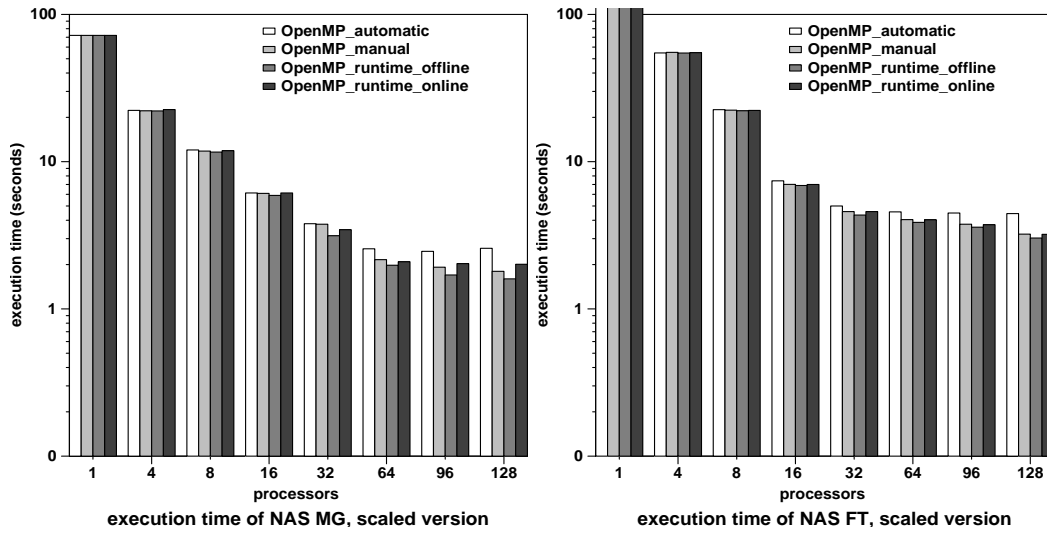


Figure 4

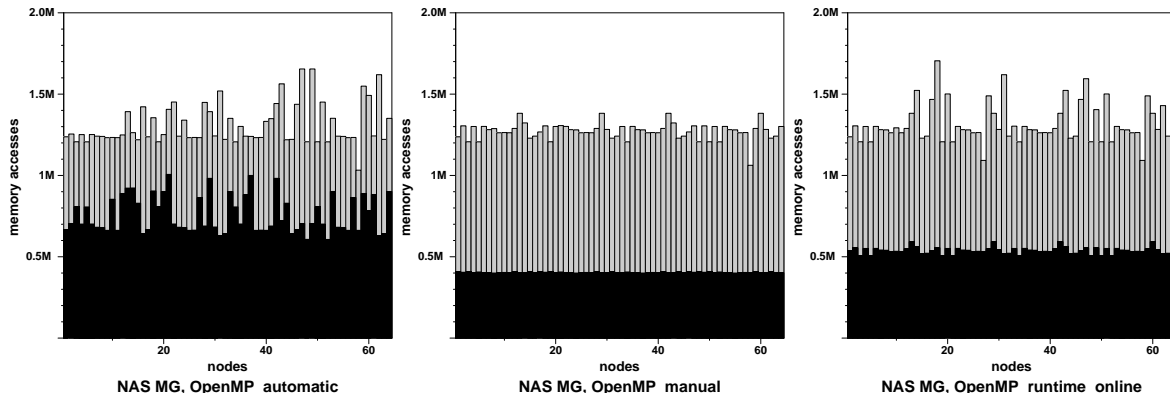


Figure 5

```

program LU
integer n
parameter (n=problem_size)
double precision a(n,n)
do k=1,n
  do m=k+1,n
    a(m,k)=a(m,k)/a(k,k)
  end do
!$OMP PARALLEL DO PRIVATE(i,j)
  do j=k+1, n
    do i=k+1,n
      a(i,j)=a(i,j)-a(i,k)*a(k,j)
    enddo
  enddo
enddo

```

```

program LU
integer n
parameter (n=problem_size)
double precision a(n,n)
!$SGI DISTRIBUTE a(*,CYCLIC)
do k=1,n
  do m=k+1,n
    a(m,k)=a(m,k)/a(k,k)
  end do
!$SGI PARALLEL DO PRIVATE(i,j)
!$SGI& AFFINITY(j)=DATA(a(i,j))
  do j=k+1, n
    do i=k+1,n
      a(i,j)=a(i,j)-a(i,k)*a(k,j)
    enddo
  enddo
enddo

```

Figure 6

```

program LU
integer n
parameter (n=problem_size)
double precision a(n,n)
integer nprocs
nprocs = OMP_GET_MAX_THREADS()
do k=1,n
  do m=k+1,n
    a(m,k)=a(m,k)/a(k,k)
  enddo
!$OMP PARALLEL DO PRIVATE(i,j,myproc,jlow),
!$OMP& SHARED(a,k)
  do myproc = 0, nprocs-1
    jlow = ((k / nprocs) * nprocs) + 1 + myproc
    if (myproc .lt. mod(k, nprocs)) jlow = jlow + nprocs
    do j=jlow,n,nprocs
      do i=k+1,n
        a(i,j) = a(i,j) - a(i,k)*a(k,j)
      enddo
    enddo
  enddo
enddo
enddo

```

Figure 7

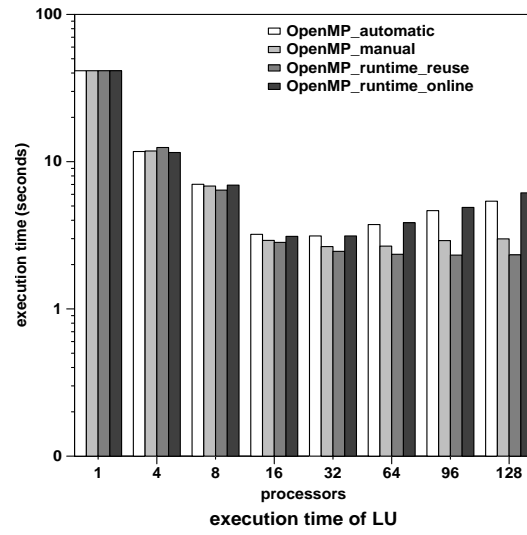


Figure 8

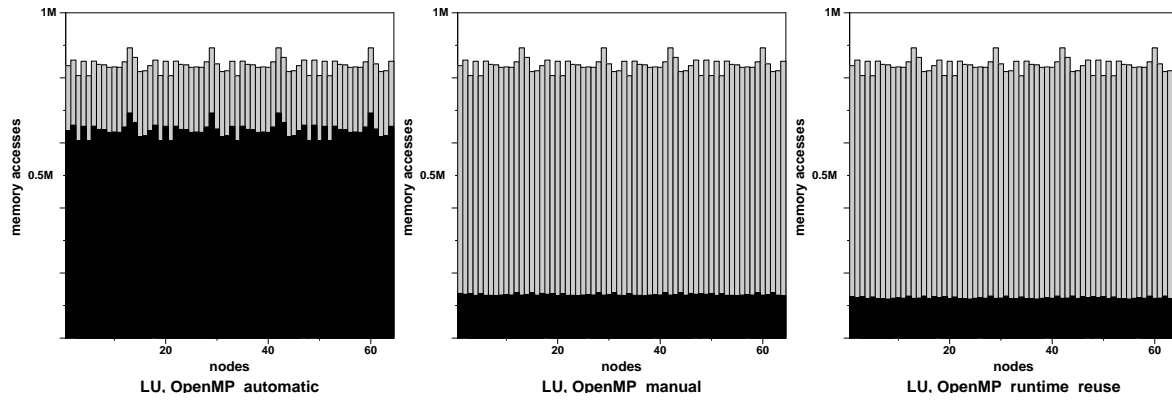


Figure 9