



The Trade-off between Implicit and Explicit Data Distribution in Shared-Memory Programming Paradigms

Dimitrios S. Nikolopoulos
Constantine D. Polychronopoulos

Coordinated Science Laboratory
University of Illinois
at Urbana-Champaign
1308 West Main Street
Urbana, IL, 61801, U.S.A.
dsn@csrd.uiuc.edu, cdp@csrd.uiuc.edu

Eduard Ayguadé
Jesús Labarta

Department d' Arquirectura
de Computadors
Universitat Politecnica de Catalunya
c/Jordi Girona 1-3
08034, Barcelona, Spain
eduard@ac.upc.es, jesus@ac.upc.es

Theodore S. Papatheodorou

Department of Computer
Engineering and Informatics
University of Patras
Rion, 26500
Patras, Greece
tsp@hpc lab.ceid.upatras.gr

ABSTRACT

This paper explores previously established and novel methods for scaling the performance of OpenMP on NUMA architectures. The spectrum of methods under investigation includes OS-level automatic page placement algorithms, dynamic page migration and manual data distribution. The trade-off that these methods face lies between performance and programming effort. Automatic page placement algorithms are transparent to the programmer, but may compromise memory access locality. Dynamic page migration is also transparent, but requires careful engineering of on-line algorithms to be effective. Manual data distribution on the other requires substantial programming effort and architecture-specific extensions to OpenMP, but may localize memory accesses in a nearly optimal manner.

The main contributions of the paper are: a classification of application characteristics, which identifies clearly the conditions under which transparent methods are both capable and sufficient for optimizing memory locality in an OpenMP program; and the use of two novel runtime techniques, runtime data distribution based on memory access traces and affinity scheduling with iteration schedule reuse, as competitive substitutes of manual data distribution in several important classes of applications.

Keywords

Data distribution, page migration, operating systems, runtime systems, performance evaluation, OpenMP.

1. INTRODUCTION

The OpenMP Application Programming Interface (API) [20] is a portable model for programming any parallel architec-

ture that provides the abstraction of a shared address space to the programmer. The target architecture may be a small-scale desktop SMP, a ccNUMA supercomputer, a cluster running distributed shared-memory (DSM) middleware or even a multithreaded processor. The purpose of OpenMP is to ease the development of portable parallel code, by enabling incremental construction of parallel programs and hiding the details of the underlying hardware/software interface from the programmer. An OpenMP program can be obtained directly from its sequential counterpart, by adding directives around parts of the code that can be parallelized. Architectural features such as the communication medium, or the mechanisms used to implement and orchestrate parallelism in software are hidden behind the OpenMP backend and do not place any burden on the programmer. The popularity of OpenMP has raised sharply during the last few years. OpenMP is now considered the *de facto* standard for programming shared-memory multiprocessors.

It is rather unfortunate that the ease of use of OpenMP comes at the expense of limited scalability on architectures with non-uniform memory access latency. Since OpenMP provides no means to control the placement of data in memory, it is often the case that data is placed in a manner that forces frequent accesses to remote memory modules through the interconnection network. Localizing memory accesses via proper data distribution is a fundamental performance optimization for NUMA architectures. Soon after realizing this problem, several researchers and vendors proposed and implemented NUMA extensions of OpenMP in two forms: data distribution directives similar to those of data parallel programming languages like HPC; and loop iteration assignment directives that establish explicit affinity relationships between threads and data [1, 2, 6, 11, 13, 16, 23].

Integrating data distribution and preserving simplicity and portability are two conflicting requirements for OpenMP. Data distribution for NUMA architectures is a platform-dependent optimization. It requires significant programming effort, usually equivalent to that of message-passing, while its effectiveness is implementation-dependent. On the other hand, injecting transparent data distribution capabilities in OpenMP without modifying the API is a challenging

Permission to make digital or hard copies of part or all of this work or personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

ICS '01 Sorrento, Italy

© ACM 2001 1-58113-410-x/01/06...\$5.00

research problem.

Recently, work from the authors has shown that at least for the popular class of iterative parallel programs with iterative reference patterns (that is, programs that repeat the same parallel computation for a number of iterations), the OpenMP runtime environment can perform implicit data distribution online, using memory reference tracing and an intelligent page migration algorithm [18]. This argument has been evaluated with a synthetic experiment, in which OpenMP programs that were manually tuned with *a priori* knowledge of the operating system's page placement algorithm, were deliberately executed with alternative page placement algorithms. The experiment has shown that no matter what the initial distribution of data is, an intelligent page migration algorithm can accurately and timely relocate pages to match the performance of the best automatic page placement algorithm. It is therefore speculated that OpenMP and shared-memory programming paradigms in general, can rely on implicit rather than explicit data distribution to achieve memory access locality.

1.1 The problems addressed in this paper

There are several circumstances in which well-tuned manual data distribution and binding of threads to memory modules provide considerably better performance than any automatic page placement algorithm [2, 3]. Given the potential of dynamic page migration as a substitute of data distribution, a vital question that remains open is whether dynamic page migration can provide the performance benefits of manual data distribution in such circumstances. A second open problem is the vulnerability of dynamic page migration to fine granularity and non-iterative or irregular reference patterns, which render online page migration algorithms incapable of localizing memory accesses [2]. Before reverting to manual data distribution in these cases, it is important to investigate whether there exist other automated procedures able to implement the same locality optimizations transparently.

The idea pursued in this paper is to replace manual data distribution and manual assignment of threads to memory modules, with a combination of runtime memory management algorithms based on dynamic page migration and loop scheduling techniques that implicitly preserve thread to memory affinity relationships. The ultimate objective is to localize memory accesses at least as well as manual distribution does, without modifying the OpenMP API.

1.2 Summary of contributions

We conducted an extensive set of experiments with the OpenMP implementations of the NAS benchmarks and a simple LU decomposition code on the Origin2000 [10]. We evaluated the native OpenMP implementations without any *a priori* knowledge of the memory management strategy of the underlying platform, thus relying solely on the page placement algorithm of the operating system for proper data distribution; the same implementations modified to use manual data distribution with HPF-like directives; and the same implementations linked with *UPMlib* [19], our runtime system which implements transparent data distribution using memory access tracing and dynamic page migration.

The NAS benchmarks are popular codes used by the parallel processing community to evaluate scalable architectures. LU is an interesting case where dynamic page migration may actually hurt performance [2] and manual data distribution combined with loop iteration assignment to memory modules appear to be the only options for scaling. In this experiment, we tested an alternative implementation of LU, in which instead of manual data distribution, the inner parallel loop is transformed to exploit memory affinity by reusing an initially computed cyclic loop schedule. This transformation ensures that in every execution of the inner parallel loop, each processor accesses the same data, thus preserving a strong thread-to-memory affinity relationship.

The findings of the paper are summarized as follows. For strictly iterative parallel codes and in situations in which data distribution appears to be necessary to improve scalability, a smart dynamic page migration engine matches or exceeds the performance of manual data distribution, under the assumption that the granularity and the memory access pattern of the program are coarse enough to provide the page migration engine with a sufficiently long time frame for tuning data placement online. In this case, dynamic page migration has several inherent advantages, most notably the ability to scan the entire memory address space instead of only distributed arrays and the ability to identify boundary cases of pages that may be misplaced with manual data distribution.

In fine-grain iterative codes, our page migration engine performs either in par or modestly worse than manual data distribution. The same happens with codes that have only a few iterations and some form of irregularity in the memory access pattern. In the first case, activating page migration or not is a matter of cost-benefit analysis, using the granularity of the parallel computation as the driving factor for taking a decision online. In the second case, the competitive algorithm of the page migration engine may fail to detect early the irregularity and incur bouncing of pages, which in turn contributes unnecessary overhead. Handling irregular access patterns with an online page migration algorithm remains an issue for further investigation.

In LU, although page migration is neither improving nor worsening performance, our findings suggest that it is possible to achieve nearly optimum memory access locality by initially assigning the iterations of the innermost loop to processors in a cyclic manner and reusing this schedule in every iteration of the outermost sequential loop. This transformation works well with a first-touch page placement algorithm and does not require manual data distribution, thread placement, or any other platform-specific OpenMP extension.

Overall, the results make a strong case for sophisticated runtime techniques that implement transparently the performance optimizations required to scale an OpenMP program on a NUMA multiprocessor. The paper contributes to a recently established belief that it is possible to use a flat directive-based programming paradigm and yet be able to obtain scalable performance, thus simplifying the process and reducing the cost of developing efficient parallel programs. We consider our work as a step towards identifying a programming model that yields the highest speedup with

the least programming effort.

1.3 The rest of this paper

The remainder of this paper is organized as follows. Section 2 gives a brief introduction to explicit and implicit data distribution mechanisms for OpenMP and reviews related work. Section 3 explains in detail our experimental setup. Our results are presented in Section 4. Section 5 summarizes our conclusions and gives some directions for future work.

2. IMPLICIT AND EXPLICIT DATA DISTRIBUTION

The idea of introducing manual data distribution directives in shared-memory programming paradigms originates in earlier work on data parallel programming languages [8, 7]. Data-parallel languages use array distributions as the starting point for the development of parallel programs. Arrays are distributed along one or more dimensions, in order to let processors perform the bulk of the parallel computation on local data and communicate rarely in a loosely synchronous manner. The communication is handled by the compiler, which has full knowledge of the data owned by each processor.

In order to use a data parallel programming model on a NUMA shared-memory multiprocessor, data distribution directives must be translated into distributions of the virtual memory pages that contain the array elements assigned to each processor. The communication is carried out directly through loads and stores in shared memory. From a performance perspective, the data parallel style of programming is attractive for NUMA architectures, because it establishes explicitly strong affinity links between computation and data.

Both researchers and vendors have implemented data parallel extensions to either platform-specific shared memory programming models, or OpenMP [2, 3, 13]. In general, 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 memory modules, the compiler needs to reorganize array subscripts in order to implement the specified mapping of elements. The second purpose of applying data parallel extensions to shared-memory programming models is to explicitly assign loop iterations to specific memory modules for exploiting 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 a potential for significant performance improvements in simple codes like LU decomposition and SOR, albeit by modifying the native programming model and sacrificing portability [2, 3, 16].

As part of an effort to provide transparent, runtime optimizations for OpenMP, we have developed a runtime system which acts as a local optimizer of memory accesses within

an OpenMP program [17, 19]. This runtime system, called *UPMlib*, monitors the reference rates from each processor to each page in memory and applies competitive page migration algorithms at specific execution points, at which the reference counters reflect accurately the exact memory access pattern of the program. In iterative parallel codes, these points are the ends of the outer iterations of the sequential loop that encloses the parallel computation. The technique is both accurate and effective in relocating early at runtime the pages that concentrate frequent 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 reference pattern, in the sense that each page is placed on the node that minimizes the latency of remote accesses [17].

Our dynamic page migration engine can potentially be used in place of manual data distribution, because monitoring of reference counters at well-defined execution points can provide the runtime system with a complete and accurate map of thread-to-memory affinity relationships. This map would otherwise be established with data distribution directives [18]. The runtime system may even identify situations in which manual data distribution places pages in a non-optimal manner, e.g. due to inability to map the whole address space of a program with distribution directives, or the absence of appropriate distributions for certain access patterns.

The results obtained so far from using *UPMlib* as a transparent data distribution tool prove that intelligent dynamic page migration algorithms can render OpenMP programs immune to the page placement algorithm of the operating system [18]. 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 used in contemporary operating systems 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 restriction of our framework is that it conforms well only to strictly iterative codes. It is questionable whether iterative codes that do not have an iterative access pattern, or non-iterative codes can benefit from dynamic page migration. In these cases, if data distribution is not an option for the sake of portability, it is more likely that careful scheduling of threads according to the location of data in memory is the most appropriate way to implement the required locality optimizations transparently.

3. EXPERIMENTAL SETUP

We conducted two sets of experiments, one using the NAS benchmarks and one using a simple hand-crafted LU decomposition. The experiments were executed on a 64-processor SGI Origin2000, with MIPS R10000 processors running at 250 MHz, 32 Kbytes of split L1 cache per processor, 4 Mbytes of unified L2 cache per processor and 12 Gbytes of DRAM memory. The experiments were conducted on an idle system, using the IRIX implementation of cpusets for runs with less than 64 processors.

3.1 NAS benchmarks

In the first set of experiments we executed five benchmarks from the NAS suite, implemented in OpenMP by researchers at NASA Ames [10]. 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 applications, while CG, MG and FT are small computational kernels. BT and SP solve the Navier-Stokes equations in three dimensions, using different factorization methods. CG approximates the smallest eigenvalue of a large sparse matrix using the conjugate-gradient method. FT computes the Fourier transform of a 3-dimensional matrix. MG computes the solution of a 3-D Poisson equation, using a V-cycle multigrid method. All codes are iterative and repeat the same parallel computation for a number of iterations that correspond to time steps.

The native OpenMP implementations of the benchmarks are tuned by their providers specifically for the Origin2000 memory hierarchy [10]. The codes include a cold-start iteration of the complete parallel computation to distribute data among processors on a first-touch basis¹ [14]. The cold-start iteration performs actually 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*) directive².

Since data distribution is already hard-coded in the benchmarks, in order to evaluate the performance of the same benchmarks without manual data distribution we explore two options. The first is to remove the cold-start iteration. Since IRIX's default page placement algorithm happens to be first-touch, IRIX eventually implements the same data placement that a manual *BLOCK* distribution would otherwise perform statically at array declaration points. The only significant difference, is that the benchmarks pay the additional cost of several TLB faults, which occur on the critical path of the parallel computation. The second option, is to use the alternative automatic page placement algorithm of IRIX, i.e. round-robin. In that case, we artificially devise a scenario in which the automatic page placement algorithm of the operating system is simply unable to perform proper data distribution. Round-robin is a realistic choice and is being used in many production settings (e.g. NCSA's large-scale Origin2000s) as the default data placement algorithm.

We executed four versions of each benchmark. In all versions, the cold-start iteration of the original code was commented out. The versions are described in the following:

- *OpenMP+rr*: This is the native OpenMP code, executed with round-robin page placement. This version corresponds to a scenario in which the programmer has absolutely no knowledge of the placement of data in memory and the OS happens to use a non-optimal

¹First-touch is the default page placement algorithm of IRIX, the Origin2000 operating system.

²This was verified experimentally, by obtaining the mappings of pages to physical memory while executing two versions of the benchmarks, one with the cold-start iteration, and one with an explicit *!\$SGI DISTRIBUTE* (*, *,*,*BLOCK*) directive at the declaration points of shared arrays.

Table 1: Data distributions applied to the OpenMP implementations of the NAS benchmarks.

Benchmark	Distributions
BT	<i>u,rhs,forcing</i> , <i>BLOCK</i> <i>z</i> -direction
SP	<i>u,rhs,forcing</i> , <i>BLOCK</i> <i>z</i> -direction
CG	<i>z,q,r,x</i> , one-dimensional <i>BLOCK</i>
FT	<i>u0, u1</i> , one-dimensional <i>BLOCK</i>
MG	<i>u,v,r</i> , one-dimensional <i>BLOCK</i>

automatic page placement algorithm. *OpenMP+rr* serves as a baseline for comparisons.

- *OpenMP+ft*: This is similar to *OpenMP+rr*, except that first-touch page placement is used instead of round-robin. In this case, the operating system happens to use the right page placement algorithm. Any additional overhead compared to an implementation with manual data distribution is attributed to TLB fault handling on the critical path of the parallel computation.
- *OpenMP+DD*: This is the native OpenMP code, extended with *!\$SGI DISTRIBUTE* directives to perform manual data distribution at declaration points of specific arrays. The pages not included in distributed arrays are distributed by IRIX on a first-touch basis.
- *OpenMP+UPMlib*: This is the native OpenMP code, executed with round-robin page placement and linked with *UPMlib* to use our page migration engine.

In the *OpenMP+DD* versions, we used *BLOCK* distributions for the array dimensions accessed by the outermost index of parallelized loop nests. This implementation is coherent with the HPF implementation of the same benchmarks, originally described in [5]. The distributions were implemented with the SGI compiler's *DISTRIBUTE* directive [3], the semantics of which are similar to the corresponding HPF distribution statement. As mentioned before, this is equivalent to using the cold-start iteration implemented in the original version of the benchmarks, combined with first-touch page placement. The only difference is that in the latter case, data is distributed in the executable part, rather than in the declaration part. We actually tested both implementations and found no difference in data distribution between the directive-based implementation and the hard-coded implementation. We emphasize that in both implementations, data distribution overhead is not on the critical path, therefore it is not reflected in the measurements.

Table 1 summarizes the data distributions applied to the benchmarks. Two benchmarks, BT and SP, might benefit from using data redistribution within iterations, in addition to data distribution. This happens because 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 *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 necessary in BT and SP, the redistributions had unacceptable overhead and poor scalability when applied in our implementations³. The results reported in the following are taken from experiments without runtime data redistribution.

In the `OpenMP+UPMlib` version, the benchmarks use the iterative page migration algorithm outlined in Section 2 and described in more detail in [17, 18]. Page migration is applied at the end of the iterations of the outer loop that encloses the parallel computation, after collecting a map of the memory accesses of the program. This algorithm performs online data distribution. Our runtime system includes also a record-replay algorithm [18], which emulates data redistribution at page-level granularity. Unfortunately, applying record-replay in BT and SP, pretty much like manual data redistribution, introduces overhead which outweighs the benefits from reducing the rate of remote memory accesses.

3.2 LU

The simple LU code shown in Figure 1 (left), is a representative example of a parallel code not easily amenable to dynamic page migration for online memory access locality optimizations. Although LU is iterative, the amount of computation performed in each iteration is progressively reduced, while the data touched by a processor may differ from iteration to iteration, depending on the algorithm that assigns iterations to processors. The iterative page migration algorithm of `UPMlib` is not of much use in this code. Neither block or cyclic data distribution can improve memory access locality significantly. A cyclic distribution along the second dimension of a is helpful though, because it distributes evenly the computation across memory modules and, implicitly, processors. In order to have both balanced load and good memory access locality, the code should be modified as shown in the right part of Figure 1. In addition to the cyclic distribution, iterations should be scheduled so that each iteration of the j loop is executed on the node where the j -th column of a is stored. On the Origin2000, this is accomplished with the `AFFINITY` directive of the SGI compiler.

We have implemented five versions of LU, using a 1400×1400 matrix⁴. Three versions, `OpenMP+ft`, `OpenMP+rr`, and `OpenMP+DD` are produced in the same manner as the corresponding implementations of the NAS benchmarks described in the Section 3.1. The `OpenMP+DD` version uses cyclic distribution along the second dimension of a , primarily for load balancing. In the same version of LU, we attempted to add the `AFFINITY` clause shown in Figure 1, in order to ensure that the j -th iteration of the innermost loop is always scheduled for execution on the node where the j -th column of a resides. Unfortunately, the SGI implementation of the `AFFINITY` clause has high synchronization overhead and flattens the speedup of LU. This phenomenon may be

³The overhead of data redistribution is most likely attributed to a non-scalable implementation on the Origin2000 [12]. Our results agree with the results in [5].

⁴We used a dense matrix merely to demonstrate the impact of data distribution.

attributed to an outmoded version of the SGI compiler we used.

The fourth version (`OpenMP+UPMlib`) uses our page migration engine, but instead of using an iterative algorithm, it uses a periodic algorithm which is invoked upon expirations of a timer [17]. The period of the algorithm is manually set to 50 ms, in order for the algorithm to have the opportunity to migrate pages within an iteration of the outermost k loop. Ideally, the mechanism would detect a change of the reference pattern at the beginning of each iteration of the k loop and migrate pages to realign data with the processors that work on the leftmost columns of the submatrix accessed during that iteration. Realistically, since it is not feasible to scan and redistribute the entire array once every 50 ms, the algorithm samples a few pages of the array whenever the timer quantum is expired.

The fifth version (shown in Figure 2) follows an alternative path for optimizing memory access locality. In this version, the inner 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 locally its own set of iterations to execute. Iterations are assigned to processors in a cyclic manner and during the k -th iteration of the outer loop, each processor executes a subset of the iterations that the same processor executed during the $(k-1)$ -th iteration of the outer loop. For example, assume that $n=1024$ and the program is executed with 4 processors. In the first iteration of the k loop, processor 0 executes iterations 2,6,10,14, ..., processor 1 executes iterations 3,7,11,15, ... and so on. In the second iteration of the k loop, processor 0 executes iterations 6,10,14, ..., processor 1 executes iterations 7,11,15 and so on. We call this version of LU `OpenMP+reuse`.

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 a first-touch page placement algorithm, such a transformation achieves good localization of memory accesses. In some sense, the transformation resembles cache-affinity loop scheduling [15]. The difference is that the transformation is used to exploit cache reuse and at the same time, localize memory accesses.

We believe that such a transformation can be relatively easy to apply for a restructuring compiler, without requiring a new OpenMP directive. In the worst case, the transformation requires an extension to the `SCHEDULE` clause of the `PARALLEL DO` directive, which dictates the compiler to compute the initial iteration schedule (cyclic in this case) and reuse it in every subsequent invocation of the loop. One way to do this, is to assign names to schedules, e.g. using a `SCHEDULE(name:policy,chunk)` clause. In the case of LU, this clause would be written as `SCHEDULE(lu_schedule:cyclic,1)`. If the compiler encounters the clause for the first time, it computes the schedule. Otherwise, in every occurrence of a `SCHEDULE(lu_reuse)` clause, the compiler identifies the precomputed schedule by its name (`lu_schedule`) and reuses it. Note that the iteration schedule reuse can be exploited across different instances of the same loop or across different loops.

<pre> 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 </pre>	<pre> 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 </pre>
---	--

Figure 1: A simple LU code implemented with plain OpenMP (left) and with data distribution and affinity scheduling (right).

```

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
!$OMP PARALLEL DO PRIVATE(i,j,mypr,jlow),
!$OMP& SHARED(a,k),
  do mypr = 0, nprocs-1
    jlow = ((k / nprocs) * nprocs) + 1 + mypr
    if (mypr .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

```

Figure 2: LU with iteration schedule reuse for expressing thread-to-memory affinity.

If the iteration schedule reuse transformation is beyond the capacity of a compiler, it can still be applied using a semi-automatic procedure. The idea is to run one iteration of the program using the *BLOCK* distribution and another iteration using the *CYCLIC* distribution and iteration schedule reuse. The two iterations are used as inspectors to *probe* the performance of the two distributions, by measuring the number of remote memory accesses to *a*. The probing run will clearly indicate the advantage of the *CYCLIC* distribution, since the number of remote memory accesses will start decreasing from the second iteration of the *k* loop.

4. RESULTS

The results from our experiments are summarized in Figures 3, 4, 5 and 6. Figure 3 shows the execution times of the benchmarks on a varying number of processors. Note that the *y*-axis in the charts is logarithmic and that the minimum and maximum values of the *y*-axis are adjusted according to the execution times of the benchmarks for the

sake of readability. Figure 4 shows the normalized speedup of the different versions of each benchmark. Normalization is performed against the execution time of the *OpenMP+rr* version. On a given number of processors, the normalized speedups are obtained by dividing the execution time of the *OpenMP+rr* version with the execution time of the other versions. Note that in two occasions, CG and LU, super-linear speedups occur between 4 and 16 processors due to working set effects. A large fraction of the capacity and conflict cache misses of the 1-processor execution is converted into hits due to additional cache space provided in the parallel execution.

Figure 5 shows histograms of memory accesses per-node, divided into local and remote memory accesses⁵ for three programs, BT, MG and LU. The selected programs are the ones that illustrate the most representative trends revealed from the experiments. The histograms are obtained from instrumented executions of the programs on 32 processors (16 nodes of the Origin2000). The instrumentation calculates the total number of local and remote memory accesses per-node, by reading the page reference counters in the entire address space of the programs.

Figure 6 shows the overhead of the page migration engine, normalized to the execution times of the benchmarks on 64 processors. This overhead is computed with timing instrumentation of *UPMlib*. It must be noted that *UPMlib* overlaps the execution of page migration with the execution of the program. The runtime system uses a thread which is invoked periodically and runs the page migration algorithm in parallel with the program. Although the overhead of page migrations is masked, the thread used by *UPMlib* interferes with one or more OpenMP threads at runtime, when the programs use all 64 processors. We conservatively assume that this interference causes a 50–50 processor sharing and we estimate the net overhead of *UPMlib* as 50% of the total CPU time consumed by the page migration thread.

We comment on the performance of the NAS benchmarks and LU in separate sections.

⁵Remote in the sense that the node is accessed by processors residing in other nodes.

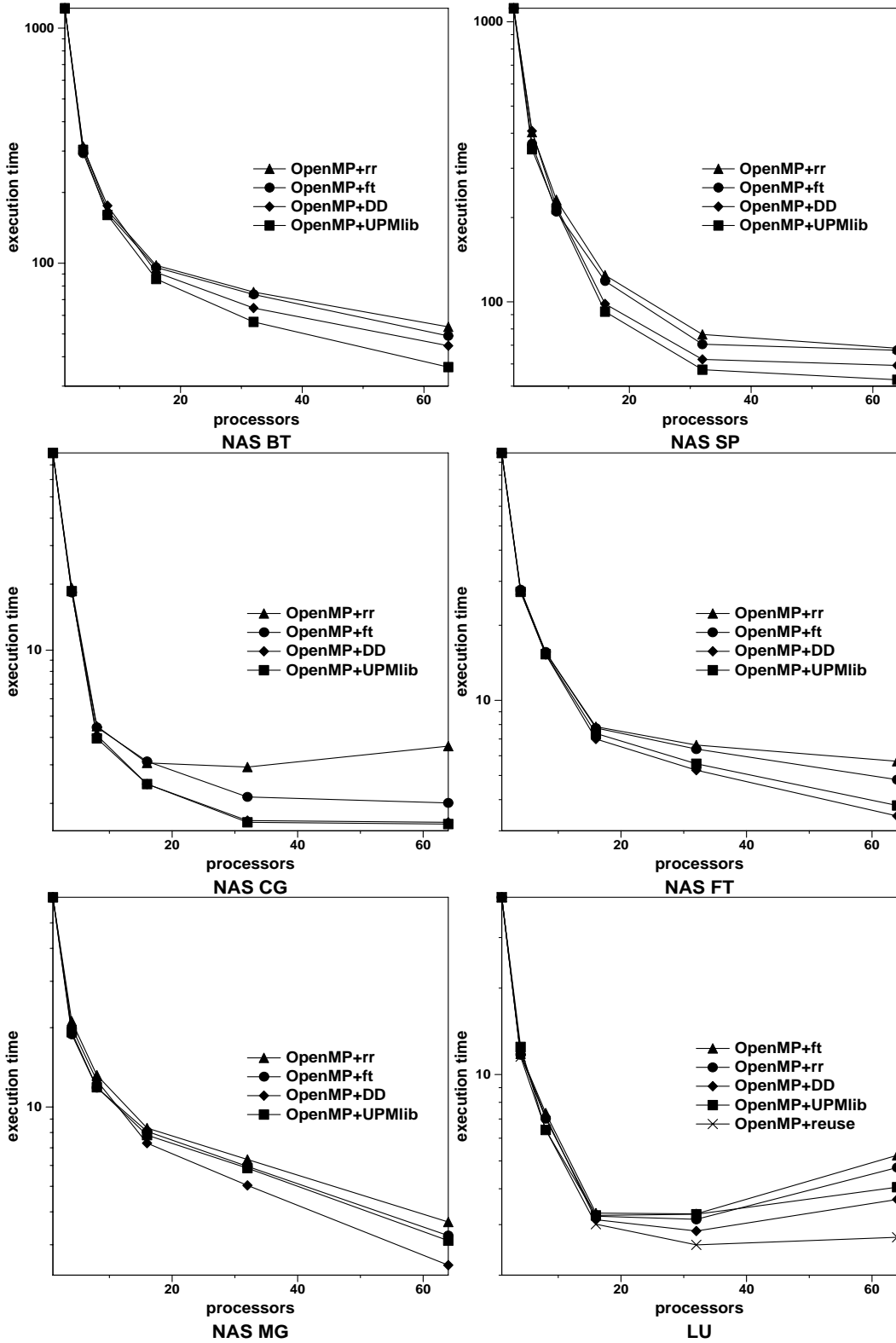


Figure 3: Execution times.

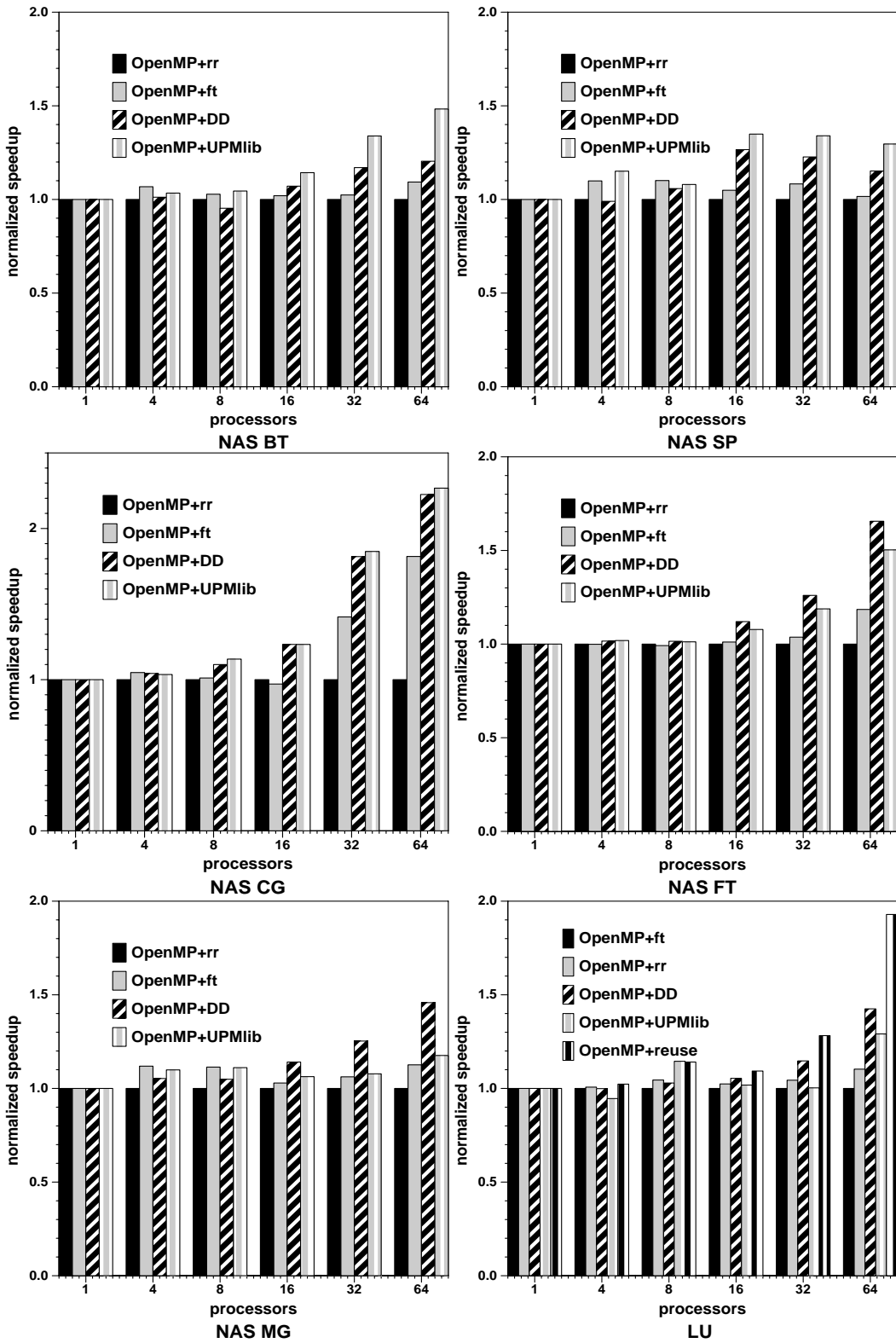


Figure 4: Normalized speedups.

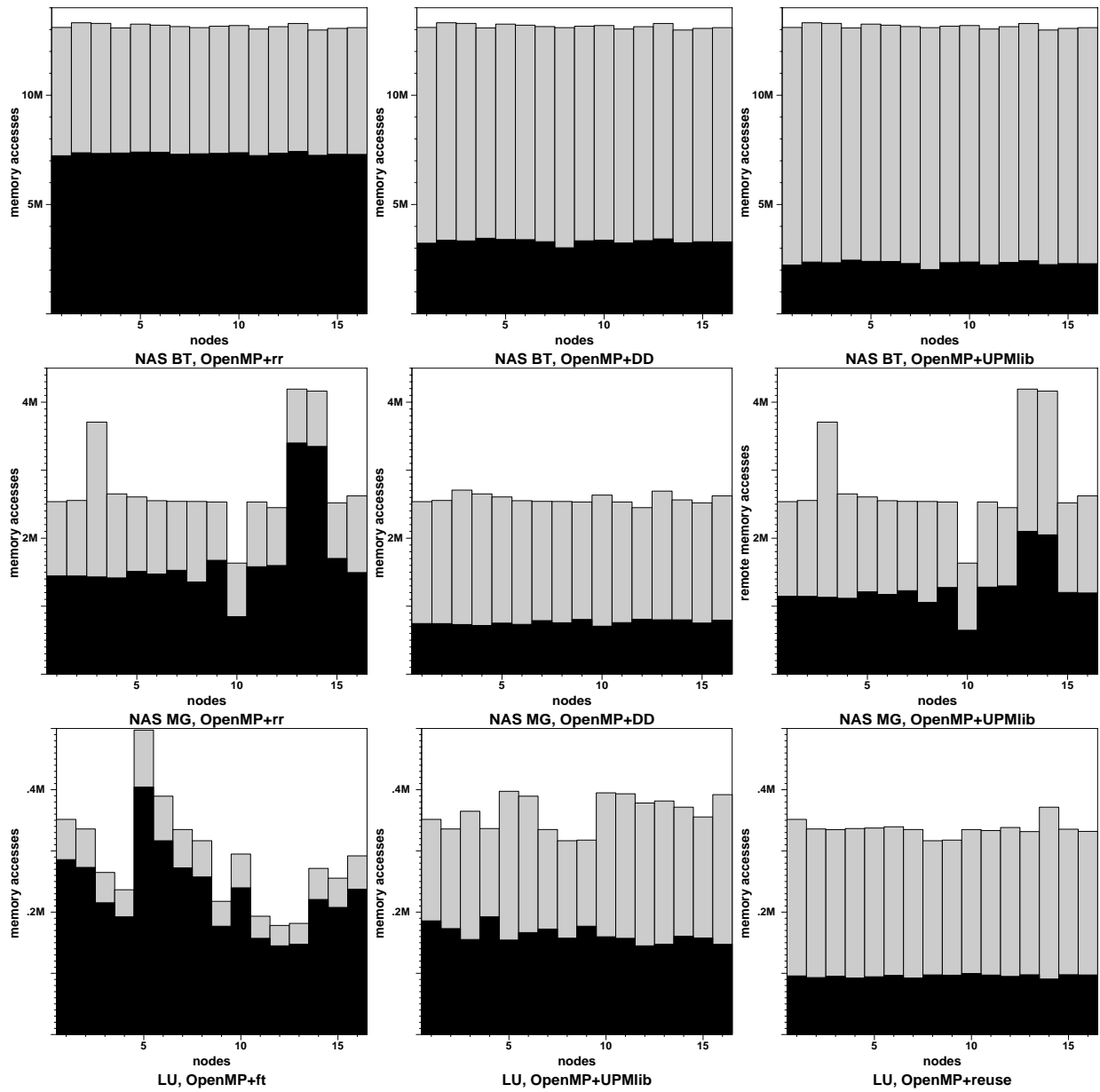


Figure 5: Per-node memory accesses of BT, MG and LU during their execution on 32 processors (16 nodes of the Origin2000). The memory accesses are divided into local (gray) and remote (black) memory accesses.

4.1 NAS benchmarks

The results from the experiments with the NAS benchmarks show a consistent trend of improvement from using manual data distribution in 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 on the Origin2000 and data placement becomes critical. On 16 processors, data distribution reduces execution time by 10%–20%. On 32 or more processors, data distribution improves performance by a wider margin, which averages 34% and in one case (CG), is as much as 81%.

The difference between manual data distribution and automatic round-robin page placement is purely a matter of the memory access locality achieved with the two distributions. Figure 5 shows that in BT, round-robin placement of pages forces 55% of memory accesses to be satisfied remotely. 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 for example, this fraction amounts to approximately 3.3 million remote memory accesses. Assuming that the base local memory access latency costs 300 ns (as stated in the system’s technical specifications) and at the 32-processor scale the remote-to-local memory access ratio is 3.3:1 (accounting for the difference between reads and writes [4]), the savings in execution time by converting all remote memory accesses to local are estimated to 2.64 seconds. The actual savings are expected to be more significant, because the localization of memory accesses reduces contention at the memory modules and the network interfaces. Contention accounts for an additional overhead of approximately 50 ns per contending node on the Origin2000 [9].

The difference between manual data distribution and first-touch placement (*OpenMP+DD* vs. *OpenMP+ft*) occurs because the former distributes pages in a blocked fashion before the beginning of the computational loop, while the latter performs the same distribution on the fly, during the execution of the first parallel iteration. The *OpenMP+ft* version pays an additional cost for TLB fault handling on the critical path of the parallel computation. *OpenMP+ft* and *OpenMP+DD* incur practically the same number of remote memory accesses (averaging 3.3 and 3.4 million per node respectively).

The behavior of our dynamic page migration engine is explained by classifying the benchmarks in two classes, coarse-grain benchmarks with regular memory access patterns and fine-grain benchmarks with potentially irregular memory access patterns.

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 a regular memory access pattern. The term *regular* refers to the distribution of memory accesses, i.e. the memory accesses, both local and remote, are uniformly distributed among the nodes. BT and SP belong to this class. Figure 5 illustrates

the regularity in the access pattern of BT. SP has a very similar memory access pattern.

In these benchmarks, our page migration engine not only matches the performance of manual data distribution, but also outperforms it by a noticeable margin, no less than 10% and in one case (BT on 64 processors), by as much as 30%. This result implies that manual data distribution does not identify accurately the complete memory access pattern of the program. It handles only pages that belong to distributed arrays. This is verified by the chart in the upper right corner of Figure 5. Compared to *OpenMP+DD*, *OpenMP+UPMlib* reduces the number of remote memory accesses per node by 31% on average.

The page migration engine can track and optimize the reference pattern of more pages than just those belonging to distributed arrays, including actively shared pages which are initially placed in the wrong node by the operating system and pages containing array elements or scalar data for which no standard distribution is adequate to improve memory access locality. The capability of accurately relocating these pages is inherent in the page migration engine, simply because the runtime system uses online information obtained from the actual execution, rather than the programmer’s understanding of the problem. This capability is useful also whenever manual data distribution misplaces pages with respect to the frequency of accesses. This can happen in boundary cases, e.g. if the array is not page-size aligned in memory and processors contend for pages at the block boundaries, or when there are pages that contain data from both distributed and non-distributed arrays or scalars. Finally, the benchmarks are coarse enough to outweigh the cost of page migration (see Figure 6) and execute enough iterations to compensate for any undesirable page migration effect, e.g. ping-pong [17].

Note that the reduction of remote memory accesses alone, is not the only factor accounting for the performance improvements on the 64-processor scale. The *OpenMP+DD* version of BT for example is about 7 seconds slower than the *OpenMP+UPMlib* version. A rough estimation of the savings from reducing the number of memory accesses yields an improvement of at most 2 seconds. We believe that further improvements are enabled by the alleviation of contention at memory modules and network links. Unfortunately, the Origin2000 lacks the hardware to quantify the effect of contention. Indirect experiments for assessing this effect are a subject of investigation.

4.1.2 Fine-grain benchmarks with potentially irregular memory access pattern

The second class includes fine-grain benchmarks with a small number of iterations and relatively short total execution time (in the order of a few seconds). CG, MG and FT belong to this class. Dynamic page migration performs in par with manual data distribution only in CG. In FT and MG, manual data distribution performs clearly better. Two reasons are likely to explain this trend. The first is the computational granularity. If the execution time per-iteration is too short, the page migration engine might not have enough time to migrate poorly placed pages in a timely manner, if at all. The second reason is the memory access pattern of

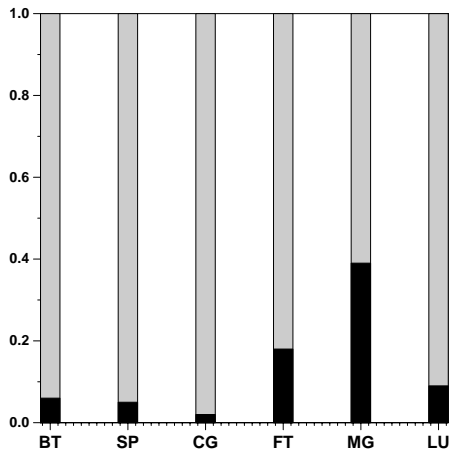


Figure 6: Relative overhead of the page migration engine (black part). The overhead is normalized to the execution time of each benchmark.

the program itself. It might be the case that due to the access pattern, the competitive page migration criterion of the runtime system causes bouncing of pages between nodes, or is simply unable to migrate pages because the reference rates do not indicate potential receivers. A node may reference a remote page more frequently than any other node, but migrating the page there may not justify the cost.

MG exposes both problems. As shown in Figure 5, MG has an irregular access pattern, in the sense that memory accesses are not distributed uniformly among the nodes. Three nodes (3, 13 and 14) appear to concentrate more memory accesses than the other nodes, while one (node 10) appears to be accessed infrequently. Manual data distribution alleviates this irregularity and results to a roughly balanced distribution of both local and remote memory accesses. Dynamic page migration on the other hand, reduces the number of remote memory accesses by 36%, but does not deal with the irregularity of the memory access pattern. The pattern remains practically unaltered, albeit with fewer remote memory accesses. The remote memory accesses of the *OpenMP+DD* version are significantly less than those of the *OpenMP+UPMlib* version. Figure 6 shows that on 64 processors, the interference between the thread that runs *UPMlib* code and the OpenMP threads accounts for 39% of the total execution time of the benchmark. A closer look at the page migration activity of MG revealed that *UPMlib* executes about 1400 unnecessary page migrations. These page migrations are for pages that simply ping-pong between nodes. Instead of reducing the number of remote memory accesses, these pages contribute a noticeable overhead which can not be masked.

To investigate further why our page migration engine is inferior to manual data distribution in MG and FT, we conducted a synthetic experiment, in which we scaled both benchmarks by doubling the amount of computation therein. We did this modification without changing the memory access pattern or the problem size of the benchmarks. We simply doubled the number of iterations of the outer sequential loop. The results from these experiments are reported in

Figure 7.

Figure 7 shows two diverging trends. In MG, *UPMlib* seems unable to match the performance of manual data distribution. In fact, the margin between the two appears to be wider in the scaled version of the benchmark. At a first glance, this result seems surprising. Timing instrumentation of *UPMlib* indicates that in the scaled version of MG, the overhead of page migration accounts for 23% of the total execution time, which is a little more than half of the overhead observed in the non-scaled version, as expected. Unfortunately, as Figure 8 shows, the fraction of remote memory accesses of MG seems to increase with page migration.

FT presents a different picture. Although there exists some bouncing of pages, increasing the number of iterations enables stability of page placement by the migration engine in the long-term. This means that page ping-pong ends early enough to be almost harmless for performance. The relative overhead of page migration in the scaled version of FT does not exceed 7% of the total execution time.

4.2 LU

We turn our interest in LU, which is the most challenging benchmark for our page migration engine. We remind the reader that a periodic timer-based algorithm is used in LU, instead of the iterative page migration algorithm used in the NAS benchmarks and that the period is selected to let the page migration engine perform at least a few migrations within a single iteration of the outer k loop. We also note that first-touch is a poor choice for automatic data distribution in LU and a cyclic distribution of pages must be used instead.

As Figures 3 and 4 show, our page migration engine does provide some measurable improvement over first-touch page placement. Figure 5 shows that in contrast to MG, *UPMlib* is able to alleviate the irregularity of the memory access pattern, which is imposed by first-touch. The reduction of remote memory accesses (more than 40%) is enough to speedup the benchmark. Nevertheless, cyclic data distribution (*OpenMP+DD*) outperforms our page migration engine, indicating that load balancing is important and that any localization of memory accesses by our page migration engine is not sufficient. 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 and reduce the impact of these shifts.

The *OpenMP+reuse* version outperforms by far the other versions, yielding an improvement of more than 50% on 64 processors. The result is encouraging, because the iteration schedule reuse transformation does not require manual data distribution. It relies on first-touch and a simple cyclic distribution of iterations to processors, which is actually computed directly from the loop bounds and therefore locally on each processor. Figure 5 suggests that the iteration schedule reuse transformation is very effective in reducing the number of remote memory accesses. This happens because each processor computes repeatedly on data that the processor touches first during the first iteration of the k loop. The transformation has also a positive effect on

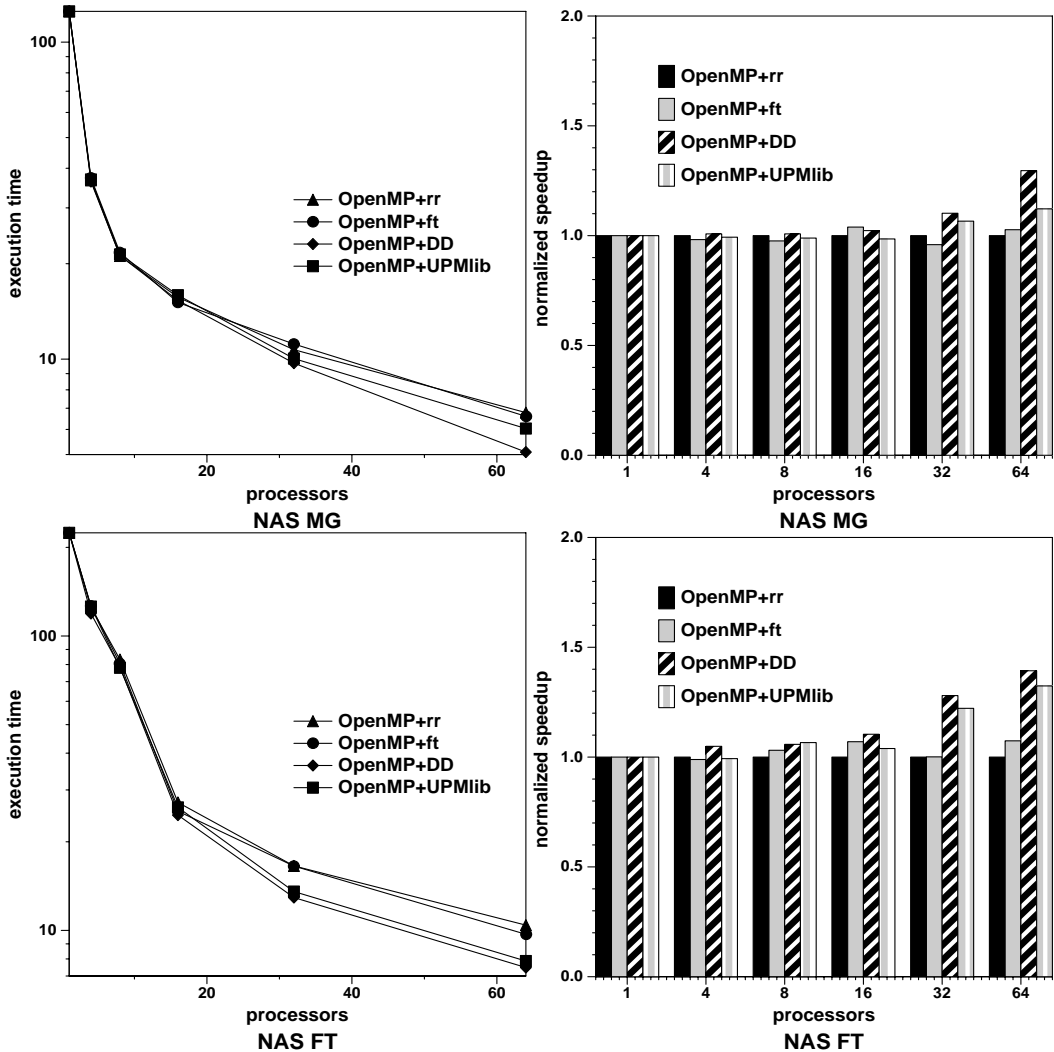


Figure 7: Execution times and normalized speedup of the scaled MG and FT benchmarks.

cache block reuse. Figure 9 shows the number of secondary cache misses and external invalidations during the execution of the various versions of LU on 64 processors. The numbers are aggregates and were collected using the IRIX *libperfex* interface to the MIPS R10000 hardware counters. It is evident that iteration schedule reuse maximizes cache block reuse, an improvement which is not accomplished by any data distribution algorithm alone. The distinctive feature of the transformation, is that it maintains an implicit binding between threads and data, which is persistent across the iterations of LU and enforces both cache and memory affinity.

The iteration schedule reuse transformation seems to be simple enough for a compiler, as soon as the compiler is able to detect that the reference pattern of LU moves along the diagonal to rectangular submatrices of progressively lower size. Since the argument of implementing this optimization in the compiler may be weak, a more realistic implementation would describe the transformation as a parameter to the *SCHEDULE* clause of the OpenMP *PARALLEL DO* construct, as outlined in Section 3. The interpretation of the

clause would be to construct a cyclic schedule for the loop and reuse it whenever the loop is executed.

We also investigated whether the combination of manual cyclic distribution and iteration schedule reuse (i.e. a *OpenMP+DD+reuse* version) improves further the scalability of LU. Such a trend was not observed in the experiments. The difference in performance between the *OpenMP+reuse* and the *OpenMP+DD+reuse* version of LU was within a range of $\pm 1.5\%$. This result is instrumental and suggests that iteration schedule reuse appears to be the only truly necessary extension to OpenMP for expressing memory affinity relationships.

5. CONCLUSIONS

We conducted a detailed evaluation of alternative methods for improving the scalability of OpenMP on NUMA multiprocessors, by localizing memory accesses. The simplest approach is to use the existing automatic page placement algorithms of contemporary operating systems, which include some NUMA-friendly policies such as first-touch. The experiments have shown that this solution is inadequate. At

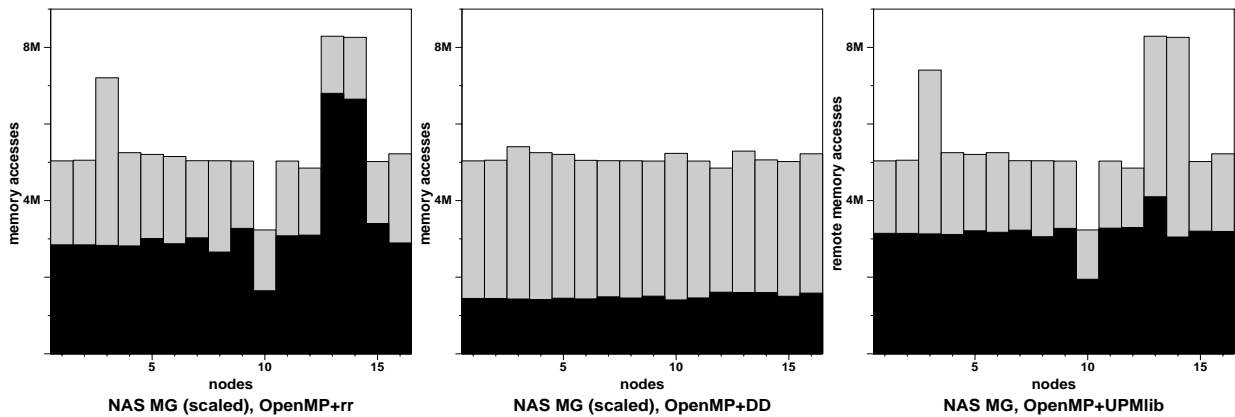


Figure 8: Per-node memory accesses of the scaled version of MG.

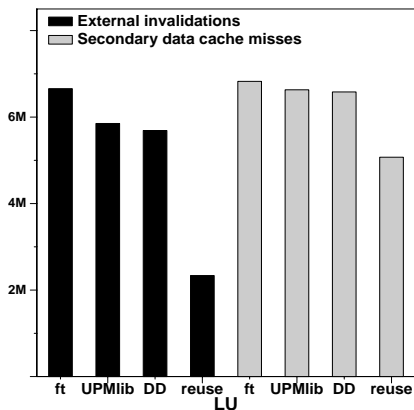


Figure 9: Accumulated L2 cache misses and external invalidations in LU.

the moderate scale of 16 processors, automatic page placement algorithms underperform static data distribution by 10%-20%, while at larger scales, their performance 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 even better performance can be obtained from a carefully engineered transparent data distribution engine, based on dynamic page migration. This is feasible under the constraints of granularity, regularity and repeatability of the memory access pattern. Page migration takes the leading edge in certain cases, because it relies on complete and accurate online information for all memory accesses, rather than on the programmer's understanding of the memory access pattern.

The constraints of granularity, repeatability and irregularity are important enough to justify deeper investigation, since several parallel codes have some or all of these properties. Our experiments with LU, a program with non-repeatable but fairly simple access pattern, gave us a hint that a balanced schedule with affinity links maintained between in-

stances of the same loop is able to localize memory accesses fairly well. This transformation requires a simple and most importantly, portable extension of the OpenMP API, rather than a full set of data distribution directives. We speculate that several codes might benefit from flexible affinity scheduling strategies that implicitly move computation close to data, rather than data close to computation. It is a matter of further investigation and experimentation to verify this intuition. We are also working on automating this procedure, using an approach similar to the well-known inspector/executor model [22]. The idea is to run a few iterations of the program in inspection mode, in order to probe the effectiveness of different data placement algorithms. In LU for example, the probing phase would alternatively test first-touch and cyclic data distribution (the latter emulated with the cyclic assignment of loop iterations to processors) and use a metric (e.g. number of remote memory accesses) to decide between the two.

The problem of granularity is hard to overcome because the overhead of page migration remains high⁶. It may worth the effort to design and implement faster hardware data copying engines on future-generation NUMA systems. As an alternative approach, we are investigating schemes for parallelizing the page migration procedure. The idea is to have each processor poll the reference 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.

The problem of dealing with irregularities in the memory access pattern remains to some extent open, in the sense that our page migration engine can at best freeze page migrations upon detecting irregularities, but is still unable to optimize the access pattern in the presence of irregularities. Irregular codes constitute a challenging domain for future work. Non-repeatable access patterns present similar problems. Adaptive codes for example, have memory access patterns which are a-priori unknown and change at runtime in

⁶The cost of moving a page on the Origin2000 is no less than 1 ms.

an unpredictable manner [24]. We are investigating solutions that combine our iterative page migration mechanism with dynamic redistribution of loop iterations and iteration schedule reuse [21]. The idea is to identify at runtime both the irregularity of the memory access pattern and potential load imbalance, and try to handle the two problems simultaneously. This can be done with a synergistic combination of user-level page migration and loop redistribution.

Acknowledgments

This work was supported in part by NSF Grant No. EIA-9975019, the Greek Secretariat of Research and Technology Grant No. 99-566 and the Spanish Ministry of Education Grant No. TIC98-511. The experiments were conducted with resources provided by the European Center for Parallelism of Barcelona (CEPBA).

6. REFERENCES

- [1] 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.
- [2] 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.
- [3] 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.
- [4] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufman, 1998.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] HPF+ Project Consortium. HPF+: Optimizing HPF for Advanced Applications. <http://www.par.univie.ac.at/project/hpf+>, 1998.
- [9] 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.
- [10] 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.
- [11] D. Kuck. OpenMP: Past and Future. In *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT'2000)*, San Diego, California, July 2000.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] 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.
- [18] 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.
- [19] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta, and E. Ayguadé. UPLib: 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.

- [20] OpenMP Architecture Review Board. OpenMP Fortran Application Programming Interface. Version 1.2, <http://www.openmp.org>, Nov. 2000.
- [21] R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime-Compilation Techniques for Data Partitioning and Communication Schedule Reuse. In *Proc. of the ACM/IEEE Supercomputing '93: High Performance Networking and Computing Conference (SC'93)*, pages 361–370, Portland, Oregon, Nov. 1993.
- [22] J. Saltz, R. Mirchandaney, and D. Baxter. Runtime Parallelization and Scheduling of Loops. In *Proc. of the 1st ACM Symposium on Parallel Algorithms and Architectures (SPAA'89)*, pages 303–312, Santa Fe, New Mexico, June 1989.
- [23] 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.
- [24] H. Shan, J. P. Singh, R. Biswas, and L. Oliker. 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.