A Study of Implicit Data Distribution Methods for OpenMP Using the SPEC Benchmarks

Dimitrios S. Nikolopoulos[1] and Eduard Ayguadé[2]

[1] Coordinated Science Lab
University of Illinois at Urbana-Champaign
1308 West Main Str., Urbana, IL, 61801
dsn@csrd.uiuc.edu

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

Abstract. In contrast to the common belief that OpenMP requires data-parallel extensions to scale well on architectures with non-uniform memory access latency, recent work has shown that it is possible to develop OpenMP programs with good levels of memory access locality, without any extension of the OpenMP API. The vehicle for localizing memory accesses transparently to the programming model, is a runtime memory manager, which uses memory access tracing and dynamic page migration to implement automatic data distribution. This paper evaluates the effectiveness of using this runtime data distribution method in non embarrassingly parallel codes, such as the SPEC benchmarks. We investigate the extent up to which sophisticated management of physical memory in the runtime system can speedup programs for which the programmer has no knowledge of the memory access pattern. Our runtime memory management algorithms improve the speedup of five SPEC benchmarks by 20–25% on average. The speedups are close to the theoretical maximum speedups for the problem sizes used and they are obtained with a minimal programming effort of about a couple of hours per benchmark.

1 Introduction

There is an ongoing debate between developers and users of OpenMP, about how should OpenMP be extended to scale better on architectures with non-uniform memory access latency, such as ccNUMA multiprocessors and clusters of SMPs [2,3,11]. Most of the related proposals converge to the conclusion that OpenMP should be extended with data distribution directives similar to the ones used in data-parallel programming languages like HPF. This argument is counterweighted by a recent research outcome [8,10] which suggests that it is possible to replace manual data distribution with intelligent runtime memory management algorithms, which infer the memory access pattern of the program

R. Eigenmann and M.J. Voss (Eds.): WOMPAT 2001, LNCS 2104, pp. 115–129, 2001.
© Springer-Verlag Berlin Heidelberg 2001

and the most appropriate placement of data from traces of memory accesses collected in hardware counters.

We have developed a runtime memory manager which localizes transparently the memory accesses of OpenMP programs on tightly-coupled ccNUMA multiprocessors [10]. The memory manager utilizes snapshots of memory access traces and dynamic page migration, to perform data distribution in a manner that minimizes the latency of remote memory accesses. The distinguishing feature of our runtime system is that it works transparently to the programmer and requires no modifications to the OpenMP API. It requires merely a simple instrumentation pass by the OpenMP translator to activate the memory manager. The runtime system detects automatically the data segment of the program and applies page migration algorithms by scanning the data segment periodically. The algorithms can exploit feedback from compiler analysis for data distribution, however they are primarily designed to operate in a fully automated process. The programming overhead for using the algorithms amounts to recompiling and linking the program with the runtime system.

The design of our runtime data distribution method is consistent with the current design principles of OpenMP, which dictate that the implementation must hide the details of the parallel architecture and the underlying hardware/software interface from the programmer. We consider our work as part of a broader effort towards building parallel programming models that meet the requirements of code and performance portability simultaneously. The purpose is to render the average programmer able to rapidly develop efficient portable parallel code and minimize the *effort/speedup* ratio, using a combination of OpenMP and runtime techniques for performance tuning.

1.1 Motivation

We have successfully applied our runtime data distribution method in parallel programs where manual data distribution would otherwise be necessary to reduce the number of remote memory accesses [7,8,9,10]. So far, we have evaluated the performance of our runtime system (named *UPMlib* after user-level page migration) using the OpenMP implementations of the NAS benchmarks [5]. Using these codes as a starting point provided us with a handful of advantages. The NAS benchmarks are highly parallel codes with coarse-grain parallelism and high sustained efficiency on most scalable parallel architectures; they are already well-tuned by their providers, the OpenMP implementations in particular are tuned specifically for the Origin2000, encompassing sophisticated cache-conscious programming and NUMA-aware data placement; moreover, the best manual data distribution algorithms, as well as the best automatic data placement algorithms for the NAS benchmarks were *a-priori* known to us, therefore we had a well-defined basis for comparisons.

In this work, we report our experience from using our runtime data distribution method with floating-point benchmarks from the SPEC CPU2000 and the SPEChpc benchmark suites [12]. Compared to the NAS benchmarks, the SPEC

Table 1. Coverage of parallel code in the SPEC benchmarks with which we experimented.

Benchmark	Coverage	Maximum speedup (Amdahl's law)
swim	77%	4.34
mgrid	80%	5.00
applu	51%	2.04
equake	93%	14.28
climate	80%	5.00

codes present us with a different picture. The native SPEC CPU2000 benchmarks are not parallelized. A very short parallelization effort of about a couple of hours per benchmark indicated that the codes expose a degree of parallelism which is well below that of the NAS benchmarks. The coverage of parallel code (ratio of the execution time of parallel code over the execution time of the entire program on a single processor) in the benchmarks we parallelized averages 76% (see Table 1). Amdahl's law suggests that the theoretical maximum speedups that the benchmarks can attain is very limited (6.13 on average). Practically, the parallelized loops are too fine-grain to provide sizable speedups. The actual maximum speedup of the hand-parallelized codes on a 64-processor Origin2000 ranges between 1.3 and 10. Of more interest to our work, is the fact that we are not aware of what is the best data placement algorithm for these benchmarks on a NUMA system, in order for us to have a point of reference for the performance of *UPMlib*.

The objective of this study is to investigate the extent up to which memory access localization can improve the scalability of codes with the characteristics of the SPEC benchmarks, on medium-scale NUMA systems. An important property common to both the SPEC and the NAS benchmarks, is that the codes are iterative, i.e. they repeat the same piece of computation for a number of iterations that correspond to ticks of a virtual timer. This is the exact class of codes for which our runtime system is more effective in localizing memory accesses [9]. Given that our runtime system is already tested against manual data distribution using iterative codes with known best data placement algorithms [7], a study with the SPEC benchmarks can provide us with an indication of the importance of memory accesses locality, without the burden of investigating, implementing and testing explicit data distribution schemes for the benchmarks.

Our findings are summarized as follows: Proper relocation of pages for localizing memory accesses has a significant impact on the scalability of the SPEC benchmarks. Linking the codes with *UPMlib* yields a speedup of up to 45% over the minimum execution time of the unmodified OpenMP code and up to 50% over the execution time with the maximum number of processors on the system with which we experimented (a 64-processor SGI Origin2000). It can be argued that the same or even higher speedup could be obtained with manual data distribution or careful restructuring of the code to localize the memory access pattern of each processor. This argument is counterweighted by the fact

that these transformations require substantially more programming effort and non-portable extensions to OpenMP.

A second outcome of our experiments is that a good fraction of the speedup obtained from our runtime page migration algorithms can be obtained with a simpler automatic page placement algorithm, which invalidates the pages that store the data accessed within parallel loops and maps them locally to each processor on a first-touch basis[1], the first time a parallel loop is executed. This mechanism resembles the generic first-touch page placement algorithm [6], but instead of being used from the beginning of execution and for the whole address space of the program, it is used right before the first iteration of the outer time-stepping loop that encapsulates the parallel computation and solely for regions of the address space which are likely to incur frequent remote memory accesses. The intuition behind the algorithm, is to place pages strictly according to the memory access pattern of the parallel code. In this way, automatic page placement is not biased by the effects of initialization, which may include sequential access of critical arrays (forcing the placement of relevant pages on a single node), or a parallel initialization phase the memory access pattern of which does not match the memory access pattern of the actual parallel computation.

The rest of this paper is organized as follows. Section 2 provides some background on the basic concepts of our runtime data distribution method. Section 3 outlines our methodology, Section 4 presents the results from our experiments and Section 5 concludes the paper.

2 Background

UPMlib uses dynamic page migration as a tool for implicit data distribution. The runtime system infers the memory access pattern of a program by taking snapshots of page reference counters [2]. The key idea of the memory management algorithms of *UPMlib* is to retrieve snapshots of reference traces that reflect accurately the memory access pattern of the whole program, or specific parts of the program for which data distribution is required to localize memory accesses. Using these snapshots, the runtime system distributes data transparently to the programmer, using a cost/benefit criterion based on the frequency and the latency of remote memory accesses to each page.

In iterative parallel codes (which constitute the majority of parallel codes in use today), implementing global data distribution with *UPMlib* is as simple as retrieving a snapshot of the memory access trace at the end of the first iteration of the parallel computation and applying the page migration criterion on this snapshot. It has been shown that this simple runtime technique performs at least

[1] The term *first-touch* refers to a page placement algorithm in which the processor that touches a page first maps the page to a local memory module.

[2] Currently, the system is implemented on the Origin2000, which collects per-node reference information for each page in hardware counters. The counters are copied back to memory by the operating system upon overflow.

Table 2. Parallelized loops in the SPEC CPU2000 benchmarks (subroutines enclosing the loops and loop labels given in parentheses).

Benchmark	Loops
171.swim	*INITAL(50,60,70,75,86)* *CALC1(100,110,115)* *CALC2(200,210,215)* *CALC3Z(400)* *CALC3(300,320,325)*
172.mgrid	*PSINV(600)* *RESID(600)* *RPJ3(100)* *INTERP(400,800)* *ZERO3(100)* *ZRAN3(400)*
173.applu	*jacld(outer k-loop)* *jacu(outer k-loop)* *rhs(outer k- and j-loops)*
183.equake	*smvp(i-loop)* *all i-loops inside main time-stepping loop*

as well and usually better than manual data distribution in coarse-grain, embarrassingly parallel codes like the NAS benchmarks [10], while it can be easily extended to implement data redistribution within iterations. Under certain circumstances, *UPMlib* outperforms data distribution, because the runtime library captures more accurate page reference information.

The main disadvantage of *UPMlib* is that it is prone to the overhead of the page migration algorithms, which require expensive operations such as data copying, TLB coherence maintenance and several system calls for accessing hardware counters. This disadvantage shows up in fine-grain codes, more specifically, when the execution time per iteration is in the order of a few tens of milliseconds or less. Note that this problem occurs in data distribution tools as well, it is not an inherent disadvantage of *UPMlib*. A second disadvantage in that the page migration algorithms of *UPMlib* require some form of repeatability in the memory access pattern. The algorithms can not handle adaptive programs, that is, codes that perform an unpredictable amount of computation in each time step. Both issues, i.e. the runtime overhead of *UPMlib* and adaptive programs are a subject of ongoing work. We attempt to address the former by parallelizing the page migration algorithms, via inlining the algorithms in the OpenMP threads. For the latter, we investigate statistical approaches for data distribution.

3 Methodology

We manually parallelized six benchmarks from the SPEC CPU2000 floating point suite, namely *swim, mgrid, equake, applu, apsi* and *lucas*. The results for *apsi*, *lucas* and *applu* are qualitatively similar. More specifically, the speedup of the benchmarks flattens beyond 2 processors, due to the very limited coverage

of parallel code (around 50%). We omit the results from the executions of these benchmarks, since they do not appear to be of interest to our study.

We parallelized the most time-consuming loops in each benchmark. We did not attempt to exploit coarse-grain task-level parallelism, although some benchmarks might benefit from it [1]. Most of the loops that we parallelized are identified as parallel by the SGI MIPSpro compiler, however in some cases, most notably in *applu* and *equake*, we had to manually apply privatization, rewriting of loop indices and induction variable elimination. The parallelized loops for each benchmark are listed in Table 2. Although the effort spent in parallelizing the benchmarks was not major, we can state with some confidence that extracting more parallelism out of these benchmarks would probably require highly sophisticated interprocedural analysis and other parallelization techniques not found in most commercial compilers. It might also require an extended execution model that exploits multiple levels of task and data parallelism [1]. Even if these techniques are applied, there is no clear evidence of their effectiveness. We have also experimented with the SPEChpc mesoscale climate modeling code. Parts of this code are already parallelized with OpenMP directives. We ran this code as distributed in the SPEChpc96 suite.

We executed three versions of the benchmarks. The first version is the unmodified parallelized OpenMP code. This code is executed using *STATIC* loop scheduling for all parallel loops and the first-touch page placement algorithm of IRIX. The IRIX kernel-level page migration engine was disabled during the experiments. IRIX includes a competitive dynamic page migration engine, which is activated by setting the *_DSM_MIGRATION* environment variable. We have run numerous tests with the IRIX page migration engine and found no case were the engine could provide meaningful improvements. All benchmarks except *swim* were slowed down by the IRIX page migration engine (*equake* slowed down by as much as 17%). *Swim*'s improvement was less than 2% on 32 processors. Note that all benchmarks except *swim* have no parallel initialization phase. Parallel initialization is helpful in certain cases, because if an automatic data placement algorithm is applied during initialization, data may be distributed quite effectively before the beginning of the main parallel computation.

The second version is the parallelized OpenMP code linked with *UPMlib*. *UPMlib* applies a competitive page migration criterion to pages accessed during the execution of parallel code at the end of the first and, if needed, subsequent iterations of the outer time-step loop [10]. During our experiments with the SPEC benchmarks, all migrated pages were identified after the execution of the first iteration and the runtime system's page migration engine was deactivated thereafter. The overhead of executing the page migration algorithm was therefore minimal. Note that although *UPMlib* minimizes its runtime overhead, the cost of page migration is still significant enough to account for, as shown in Section 4.

The third version of the benchmarks is produced from the OpenMP code with the following modification. The pages accessed during the execution of parallel code are invalidated with the *mprotect()* system call, right before the first iteration of the outer time-stepping loop. We install a handler for the SIGSEGV

signal, which records the address of the faulting page and maps the page to a local frame (i.e. residing on the same node with the processor that incurs the fault) using *mmap()*. This modification implements a restricted form of first-touch page placement. In particular, pages are mapped locally to the processor that touches them first during the first executed instance of a parallel loop. This ensures that pages are placed on a first-touch basis, according to the page reference pattern of the parallelized part of the code, rather than the reference pattern of the program as a whole. Basically, the mechanism discards any inopportune placement of pages that might be performed from the operating system during the initialization phase. We applied one optimization in the algorithm, for situations where the memory access patterns of different parallel loops do not match. In these cases, the algorithm applies first-touch page placement during the most time consuming loops with the same access pattern. We applied the algorithms in all benchmarks except *swim*. *Swim* already includes a parallelized initialization phase, which performs implicitly the placement of pages that our modification would otherwise do during the first iteration of the time-stepping loop.

Our hardware platform is a 64-processor (32-node) SGI Origin2000, with MIPS R10k processors running at 250 MHz. Each processor has 32 Kbytes of split L1 cache and 4 Mbytes of unified L2 cache. The system has 12 Gbytes of DRAM memory. The experiments were conducted on an idle system. The reported execution times are medians of three runs. We used the train problem sizes. Although this was mandated by system administration restrictions in the time available for running experiments on an idle system, the selected problem sizes are coarse enough to indicate parallel performance trends[3]. The maximum number of processors used was 62, because we detected contention between the IRIX kernel and the application threads when the benchmarks used all 64 processors. The codes were compiled the with -O2 optimization level.

4 Results

Figure 1 illustrates the execution times of four benchmarks versus the number of processors. Note that the y-axes are drawn logarithmic and the minimum and maximum values of the axes are tuned according to each benchmark's parallel execution time, for the sake of readability. The labels correspond to the unmodified OpenMP code (*OpenMP*), the OpenMP code linked with *UPMlib* (*OpenMP+upmlib*) and the OpenMP code modified to use our optimized first-touch page placement algorithm (*OpenMP+opt_ft*).

We observe a clear trend in the results. *UPMlib* reduces the minimum execution time of the benchmarks, regardless of the number of processors on which this time is obtained, by up to 45% (see Table 3). On the maximum number of pro-

[3] The granularity of the train problem size is roughly equivalent to that of the Class A problem size of the NAS benchmarks. The execution time per iteration is in the order of several hundreds of milliseconds.

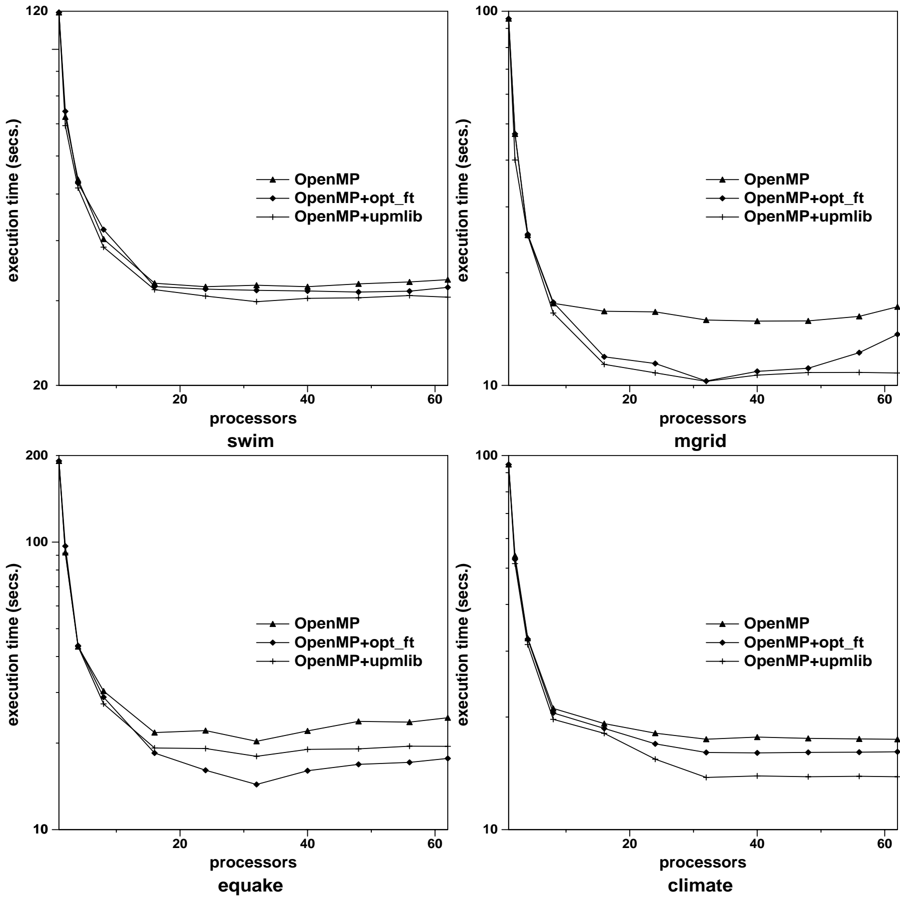

Fig. 1. Execution times.

cessors the improvement is up to 50%. On average, the margin of improvement is similar to that observed for the NAS benchmarks [7].

Figures 2 and 3 illustrate how *UPMlib* localizes memory accesses for higher performance. The charts show histograms of memory accesses from the executions of the benchmarks on 32 processors, divided into local (gray part) and remote (black part) memory accesses. The applied page migration algorithms have two important effects. First, they convert a significant fraction of remote memory accesses (in all cases more than 50%) into local memory accesses. On the Origin2000, this translates into a net saving of at least 100 ns. and up to 800 ns. per memory access [4]. The reductions in execution time are roughly proportional to the amount of remote memory accesses converted into local ones by *UPMlib*. The benchmarks with more remote memory accesses benefit more from dynamic page migration. *Mgrid* and *climate*, which average 1.5 and 6 million remote memory accesses per node, enjoy speedups of 45% and 26% respectively.

Table 3. Reduction of execution time (in percent) with *UPMlib* and our optimized first-touch algorithm.

	UPMlib		*Optimized first touch*	
	min. exec. time	*exec. time on 62 proc.*	*min. exec. time*	*exec. time on 62 proc.*
swim	*-7.4*	*-8.7*	*-2.5*	*-3.7*
mgrid	*-44.8*	*-50.3*	*-44.5*	*-18.4*
equake	*-12.7*	*-25.7*	*-41.2*	*-38.4*
climate	*-26.4*	*-25.9*	*-8.7*	*-8.0*
avg.	***-20.3***	***-27.7***	***-24.2***	***-17.1***
stdev	***16.7***	***32.4***	***29.7***	***20.5***

The second and apparently equally important effect of our runtime data distribution method is the alleviation of contention. The memory access traces reveal that all benchmarks have a highly unbalanced pattern of remote memory accesses. This means that a few nodes are accessed remotely significantly more frequently compared to the other nodes. The nodes that concentrate frequent remote memory accesses are likely to suffer from contention at their memory modules and network links. Contention is an important, yet underestimated effect on the performance of NUMA systems. On the Origin2000, the contention factor may account for as much as an additional 50 ns. per contended node per remote memory access [4]. *UPMlib* reduces contention by reducing and distributing evenly the remote memory accesses. The right charts in Figures 2 and 3 show that *UPMlib* achieves an almost perfectly balanced distribution of remote memory access in *mgrid* and *equake*. The remote memory accesses in *swim* are lightly unbalanced, however the low number of remote memory accesses per processor makes the contention effect almost imperceptible.

Intuitively, we expected somewhat more improvements, because unlike the OpenMP implementations of the NAS benchmarks [5], the SPEC codes are not tuned for efficient execution on a NUMA system. Practically, this effect does not show up in the experiments because the scalability of the SPEC benchmarks is primarily limited by the limited coverage of parallel code. A closer look at the results reveals that the benchmarks that have the higher speedup are also the ones that benefit more from the use of our page migration engine. Based on this observation, we speculate that extracting more parallelism out of the benchmarks is likely to make the impact of memory access localization more profound and the use of our runtime data distribution method vital. On the other hand, the magnitude of difference between the *OpenMP* and the *OpenMP+upmlib* versions indicates that the hardware of the Origin2000 is quite effective in reducing the impact of remote memory accesses, by guaranteeing a relatively low remote-to-local memory access latency ratio. We expect this trend to prevail in next-generation NUMA systems, which include full-fledged hardware mechanisms for reducing the impact of remote memory accesses, such as remote access caches and COMA protocols.

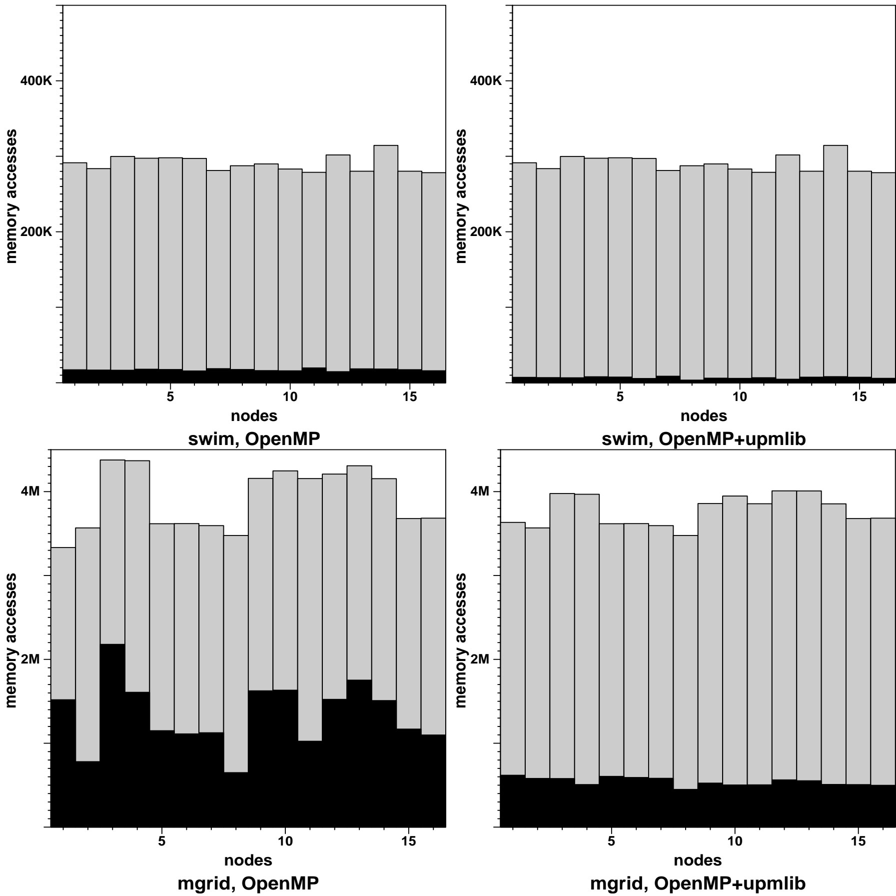

Fig. 2. Memory access histograms of *swim* and *mgrid* during their execution on 32 processors (16 nodes) of the Origin2000.

Interestingly, our optimized first-touch algorithm outperforms the native IRIX page placement algorithm. In two benchmarks, *swim* and *climate*, the improvements are considerably lower compared to those yielded by *UPMlib* (see Table 3). Figure 4 shows that in *climate*, the optimized first-touch algorithm has more remote memory accesses. The pattern of remote memory accesses is also highly unbalanced, which is a strong indicator of contention. The results show that first-touch is not the best choice of an automatic page placement algorithm for *climate*. We attempted to fix this problem using a round-robin page placement algorithm instead of first-touch without success. Round-robin performs slightly better than first-touch because it distributes better the remote accesses. However, the actual number of remote memory accesses is increased with round-robin. The memory access pattern of *climate* is both unbalanced

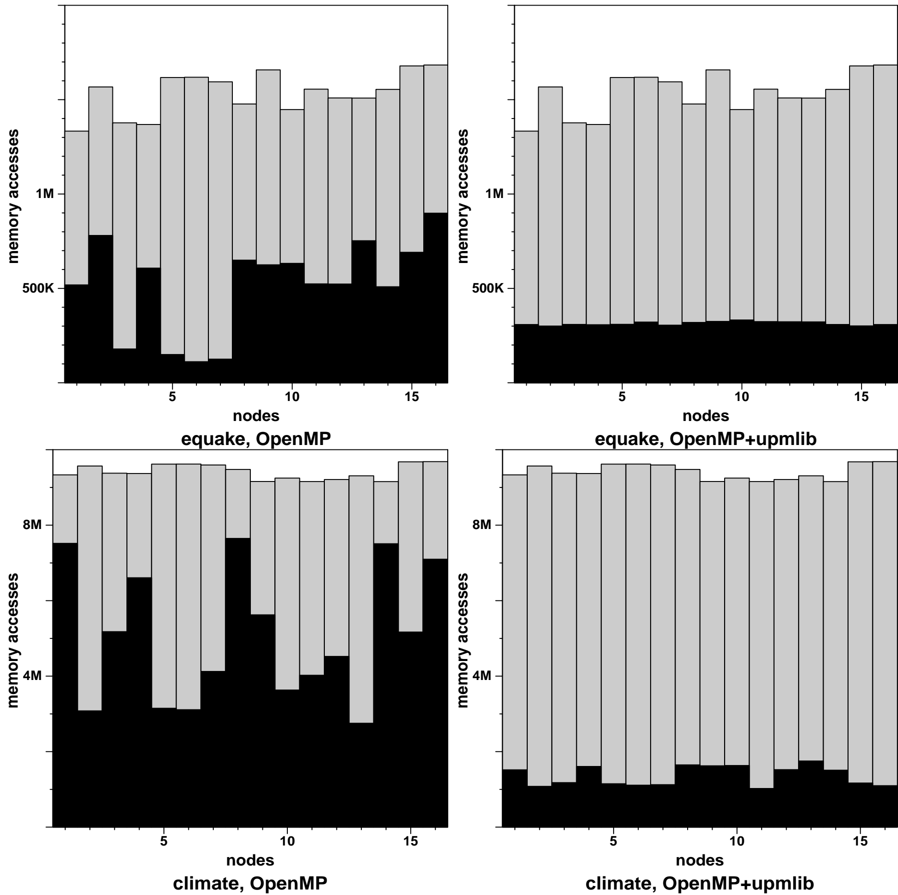

Fig. 3. Memory access histograms of *equake* and *climate* during their execution on 32 processors (16 nodes) of the Origin2000.

and irregular, therefore dynamic page migration is the best option for optimizing memory access locality.

Both the page migration algorithm and the optimized first-touch algorithm perform approximately the same in *mgrid*. In *equake*, the optimized first-touch algorithm outperforms the IRIX page placement algorithm by a significantly wider margin compared to our dynamic page migration engine. This result is somewhat surprising. Figure 4 shows that the optimized first-touch algorithm incurs less remote memory accesses than the page migration algorithm. We were not able to find a convincing explanation for this effect, other than that *UPMlib* relocates pages that concentrate frequent remote memory accesses later than the optimized first-touch algorithm. The reason which is more likely to explain the performance of the page migration engine in *equake* is the overhead of page migrations.

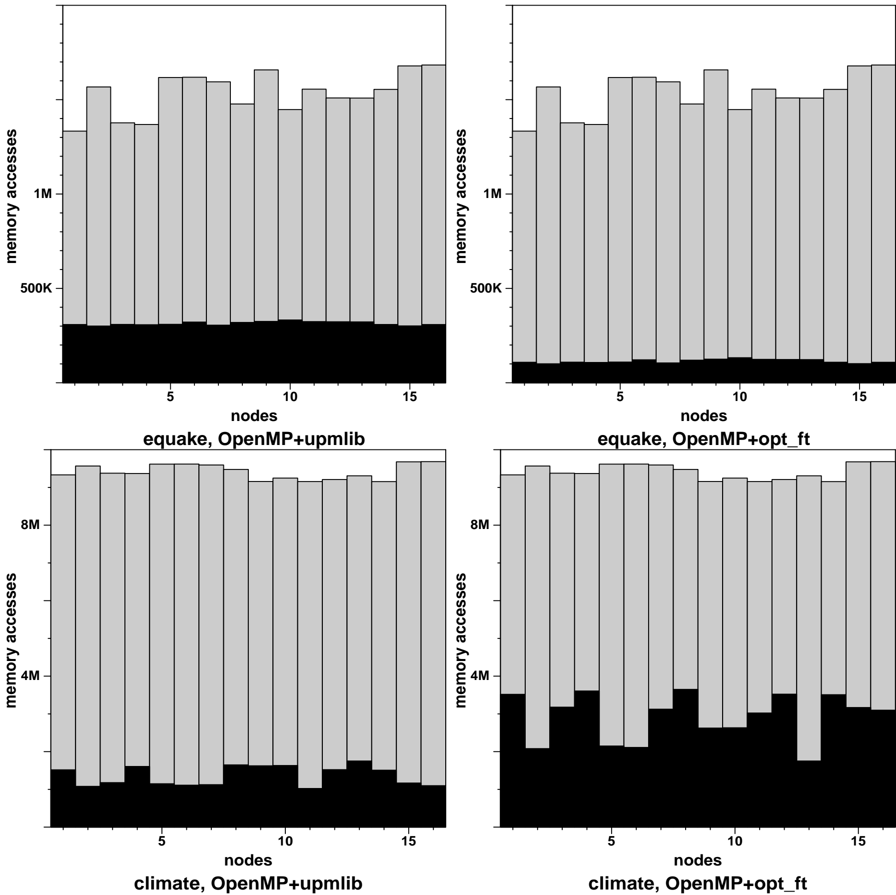

Fig. 4. Comparison of the memory access traces of the page migration engine and the optimized first-touch algorithm in *climate* and *equake*.

Figure 5 shows the relative overhead of the page migration algorithm during the executions of the benchmarks on 62 processors. *UPMlib* uses a thread that executes the page migration algorithms in parallel with the program. Although the overhead of page migration is overlapped, there is still an interference between the *UPMlib* thread and one or more *OpenMP* threads. We conservatively estimated this interference by measuring the CPU time spent by the *UPMlib* thread and assuming that 50% of this CPU time is spared from a thread of the program. The relative overhead of page migration in *equake* exceeds 20% of the execution time of the benchmark. As a comparison, the relative overhead of the optimized first-touch algorithm in *equake* is 4% of the total execution time (shown in the right chart of Figure 5). The overhead of page migration is noticeable in *swim* and *climate* as well.

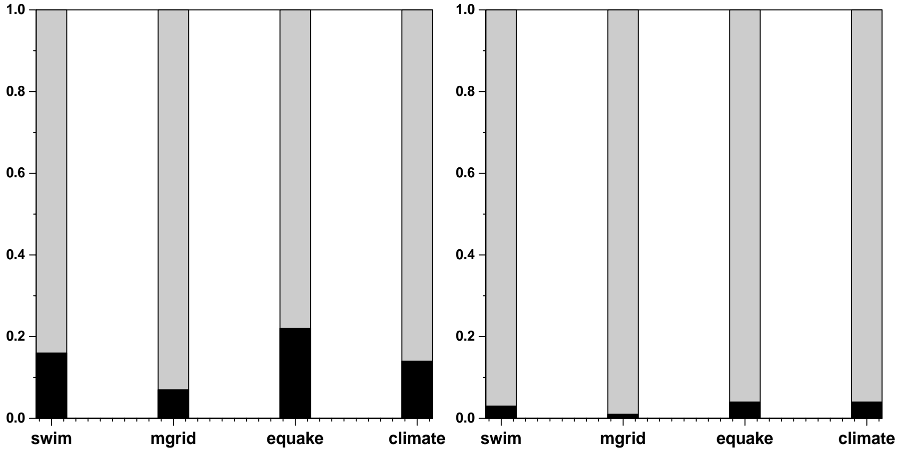

Fig. 5. Overhead of page migration (left) and the optimized first-touch algorithm (right), normalized to the execution time of the benchmarks.

The performance of our optimized first-touch algorithm brings life to an old idea suggesting that automatic page placement algorithms for NUMA systems can be more effective, when coupled with hints from the runtime system about the points of execution at which the algorithms should be activated (referred to as phase changes in the related literature [6]). The actual contribution to our OpenMP memory management framework is a low-cost, transparent mechanism for localizing memory accesses, which might prove of practical use in cases where dynamic page migration is vulnerable, most notably in fine-grain parallel codes [7].

5 Conclusions

This paper analyzed the performance of runtime memory management algorithms that localize the memory accesses of OpenMP programs on a NUMA system, using programs from the SPEC benchmark suites. The SPEC benchmarks are challenging for automatic memory management algorithms, because they are not embarrassingly parallel, neither are they tuned for efficient execution on NUMA systems. Linking the codes with our runtime system yielded a solid performance improvement of 20–25% on average. Similar or somewhat lower improvements were obtained from a simpler algorithm that places pages on a first-touch basis during the first invocation of each parallel loop. This mechanism may be valuable when the overhead of page migration is non-negligible. Overall, the results are consistent with a recently established trend that favors the use of intelligent runtime methods to scale OpenMP on clustered NUMA architectures, without modifying the appealing OpenMP API [9]. We plan to take several steps in this direction, hoping to secure performance portability with unmodified implementations of the OpenMP standard.

Acknowledgments. Jesús Labarta, Theodore Papatheodorou and Constantine Polychronopoulos have contributed valuable insight in earlier stages of this research. This work was supported by NSF Grant No. EIA-9975019 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).

References

1. E. Ayguadé, X. Martorell, J. Labarta, M. González, and N. Navarro. Exploiting Multiple Levels of Parallelism in OpenMP: A Case Study. In *Proc. of the 1999 International Conference on Parallel Processing (ICPP'99)*, pages 172–180, Aizu, Japan, August 1999.
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, August 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, November 2000.
4. D. Lenoski C. Hristea 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, November 1997.
5. 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, October 1999.
6. 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, April 1995.
7. D. Nikolopoulos, E. Ayguadé, J. Labarta, T. Papatheodorou, and C. Polychronopoulos. The Trade-Off between Implicit and Explicit Data Distribution in Shared-Memory Programming Paradigms. In *Proc. of the 15th ACM International Conference on Supercomputing*, Sorrento, Italy, June 2001.
8. D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta, and E. Ayguadé. A Transparent Runtime Data Distribution Engine for OpenMP. *Scientific Programming*, May 2001.
9. 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, November 2000.
10. D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta, and E. Ayguadé. UPMlib: 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.

11. 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.
12. Standard Performance Evaluation Corporation (SPEC). SPEC CPU2000 and SPEC hpc96 documentation. http://www.spec.org, accessed 2001.