Leveraging Transparent Data Distribution in OpenMP via User-Level Dynamic Page Migration

Dimitrios S. Nikolopoulos[1], Theodore S. Papatheodorou[1]
Constantine D. Polychronopoulos[2], Jesús Labarta[3], and Eduard Ayguadé[3]

[1] High Performance Information Systems Laboratory
Department of Computer Engineering and Informatics
University of Patras, Greece

[2] Center for Supercomputing Research and Development
Coordinated Sciences Laboratory
University of Illinois at Urbana-Champaign

[3] European Center for Parallelism of Barcelona
Polytechnic University of Catalonia, Spain

Abstract

This paper describes the mechanisms offered by UPMLIB for emulating data distribution and redistribution, features included in data parallel programming paradigms (for instance, High Performance Fortran). UPMLIB allows the compiler to insert in the application, without neither programmer intervention nor modification of OpenMP, a smart user-level page migration engine that makes it immune to performance flaws related to page placement. This engine can accurately and timely fix poor initial page placement schemes. However, our dynamic page migration engine is unable to tune page placement at a fine-grain time scale, such as in cases in which the program exhibits phase changes in the page reference pattern. The effectiveness of page migration in these cases is limited by the overhead of coherent page movements and the low remote-to-local memory access latency ratio of contemporary ccNUMA systems.

1 Introduction

The OpenMP API [11] offers a simple and flexible interface for programming parallel applications on shared-memory multiprocessors. OpenMP has recently attracted major interest from both the industry and the academia, due to two strong inherent advantages: portability and simplicity. OpenMP is portable across a wide range of shared-memory platforms, including desktop SMPs, small to medium-scale bus-based multiprocessors, scalable cache-coherent NUMA systems and most recently, clusters of workstations and SMPs [3, 11]. The simplicity is mainly provided by the use of a directive-based approach and a simple fork/join model of parallel execution. The programmer annotates sequential code with directives that enclose loops or blocks of code that can be executed in parallel. It offers an intuitive, incremental approach for developing parallel programs and follows the simple fork/join model of parallel execution. Parallel constructs are executed by a group of threads, which is controlled by a master thread. The group of threads is transparently scheduled on multiple physical processors via customized runtime and OS support.

The most prevalent problem of OpenMP is that simplicity is often traded for performance. OpenMP programs scale difficultly to tens or hundreds of processors. Some researchers have pinpointed this effect as a problem of the overhead of managing parallelism in OpenMP, which poses constrains on the granularity of parallelism that can be exploited at large processor scales [5]. Although one could argue that this is a problem of shared-memory programming models in general, it has been shown that programs parallelized with the shared-memory communication paradigm can achieve satisfactory scaling up to a few hundreds of processors with relatively reasonable programmer interventions, the most important of which is proper data distribution among processing nodes [4]. What appears to be the true problem of OpenMP is the poor interaction between the parallelized code and the deep memory hierarchies of contemporary shared-memory multiprocessors.

OpenMP provides no means to the programmer for controlling the distribution of data among processors. On scalable shared-memory systems with non-uniform memory access latencies, the distribution of data can have a significant impact on the performance of memory-intensive programs, if pages with shared data are distant from the threads that access them more frequently upon cache misses. To surmount this problem, vendors provide either data distribution directives as extensions to OpenMP or operating system support to control the placement [1] and dynamic migration [13] of data pages.

Offering data distribution directives similar to the ones offered by High-performance Fortran (HPF [6] has two fundamental shortcomings. First, it is inherently platform-dependent and thus hard to standardize and incorporate seamlessly in shared-memory programming models like OpenMP. Second, it is subtle for programmers and compromises to a significant extent the simplicity of shared-memory programming models, thus weakening one of the strongest advantages of OpenMP. Basically, these extensions are oriented towards offering facilities to express alignment (`ALIGN`: how the different dimensions of the arrays are related) and distribution (`DISTRIBUTE`: dimensions of the arrays are distributed among the processors available in a `BLOCK`, `CYCLIC` or `BLOCK-CYCLIC` way). In large programs where different computationally intensive phases occur, the `REALIGN` and `REDISTRIBUTE` directives can be used to dynamically modify the initial data mapping.

Dynamic page migration, triggered by specialized hardware that monitors the reference rates of from each node to each page in memory, moves pages competitively between nodes, based on the observed reference rates [13]. Page migration can be employed as an optimization for programs with dynamically changing reference patterns, but also, as a system tool to automatically fix incorrect placements of pages at runtime. Though transparent to the programmer, requires complicated policies and algorithms in order to meet both application and system resource management requirements; it is hard and expensive to implement in real operating systems; and has yet to demonstrate the expected results on actual systems, despite the promising initial results which were based on simulation [4, 13].

In this paper we present an integrated compiler/runtime/OS framework for emulating data dis-

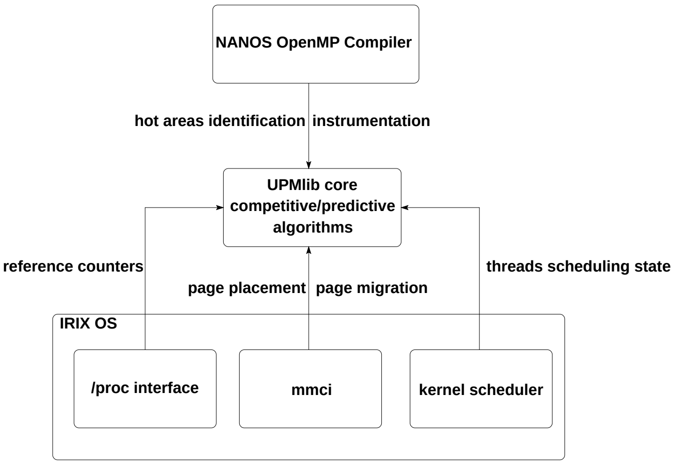

Figure 1: UPMLIB modules and interfaces.

tribution capabilities without modifying OpenMP. Section 2 overviews the user–level dynamic page migration scheme implemented in UPMLIB. Section 3 provides details on how the migration engine is inserted by the compiler in the parallel code in order to emulate data distribution and redistribution. Section 4 provides a set of experimental results that substantiate our arguments for dynamic page migration as a substitute for page distribution and redistribution in OpenMP. Finally, Section 5 concludes the paper.

2 An Overview of User-Level Dynamic Page Migration

We have recently presented a framework for user-level dynamic page migration on ccNUMA systems, implemented with the IRIX 6.5 operating system, for Origin2000 multiprocessors [8]. The key design issue of this framework is the integration of the OpenMP compiler and the operating system with the page migration engine, in order to improve the accuracy and timeliness of page migrations, as well as amortize the cost of page migration better, compared to page migration engines hardwired in the operating system.

Figure 1 illustartes the architecture of our user-level page migration engine, called UPMLIB. The OpenMP compiler identifies *hot* memory areas of the application's virtual address space which are likely to contain pages eligible for migration, and instruments the program to invoke the user-level page migration engine at specific points of execution, or periodically, depending on the semantics of the parallel computation.

The core of *UPMlib* includes a handful of algorithms for page migration, with different degrees of aggressiveness, sophistication, and adaptability. In general, the runtime system applies a competitive page migration algorithm, which takes into account the following factors: the dynamic page reference pattern of the program, obtained via monitoring of the Origin2000 hardware memory reference counters; the non-uniformity of memory access latencies in the system; contention; and the effectiveness of the algorithm itself, which is monitored at runtime by measuring the rates of remote memory references and the page migration activity, in order to embed a self-adjustment of the tunable page migration parameters in the page migration engine.

Competitive algorithms are used for standalone parallel programs, to tune the placement of pages across the nodes of the system according to the characteristics of the program. Aggressive page migration algorithms, such as predictive algorithms and page forwarding [8, 10], are used to cope with the effects of multiprogramming, which poses frequent migrations of threads between the nodes of the system. The aggressive page migration algorithms exploit scheduling information provided by the operating system with respect to the instantaneous mapping of threads to processors and trigger page forwarding mechanisms upon intercepting thread migrations. The page forwarding mechanisms discard any obsolete page reference history and correlate the reference counting information with the actual scheduling status of the computation to infer the node for which each page has affinity after the threads that used to access the page more frequently have moved.

UPMLIB uses essentially three low-level system services, namely the `/proc` interface for accessing the hardware and software-extended page reference counters of the Origin2000; the IRIX memory management control interface (`mmci`) for virtualizing the physical memory of the system via Memory Locality Domains (MLDs) and implementing page migrations between MLDs associated with physical nodes of the system; and the IRIX `schedctl` interface, for efficiently exchanging scheduling information with the IRIX kernel [12].

3 Implementing Data Distribution facilities at User-Level

This section discusses the mechanisms implemented in UPMLIB for emulating data distribution, redistribution and hybrid schemes in OpenMP programs, without intervention from the programmer.

3.1 Data Distribution

UPMlib approximates the desired initial data distribution of a program by eagerly migrating pages at the beginning of the execution, to ensure stable memory performance for the program in the long-term. The fundamental idea, is to try to move all pages to the appropriate nodes using information that is available early in the execution of the program. The page migration engine uses two mechanisms for this purpose. The first mechanism is designed for iterative programs, i.e. programs that enclose the complete parallel computation in an outer sequential loop. The vast majority of parallel codes, including the popular programs for benchmarking parallel systems (such as SPECfp and NAS), belong to this category. The second mechanism is designed for non-iterative programs and programs which are iterative, but do not repeat the same reference pattern in every iteration.

In order to activate the iterative mechanism, the OpenMP compiler instruments the programs to invoke the page migration engine at the end of every outer iteration of the computation. At these

```
...
call upmlib_init()
call upmlib_memrefcnt(u, size)
call upmlib_memrefcnt(rhs,size)
call upmlib_memrefcnt(forcing,size)
...
do step=1,niter
    call compute_rhs
    call x_solve
    call y_solve
    call z_solve
    call add
    call upmlib_migrate_memory()
enddo
```

Figure 2: An example of the usage of the UPMlib iterative page migration mechanism in the NAS BT benchmark.

points, the page migration engine can obtain an accurate snapshot of the complete page reference pattern of the program, and optimize page placement with respect to this snapshot. In most cases, it is sufficient to obtain the snapshot after the execution of the first iteration of the parallel program to place all the pages at the appropriate nodes, i.e. place each page so that the maximum latency due to remote accesses by any node to this page is minimized. Snapshots from more than one iterations are needed in cases in which some pages are ping-ponged between more than one nodes due to page-level false sharing. In these cases, *UPMlib* freezes the ping-ponging page after one migration. Migrating pages at the end of outer iterations improves the accuracy of page migration decisions, while moving all pages within the first couple of iterations of the program ensures the timeliness of page migrations. Figure 2 gives an example of the usage of the iterative page migration mechanism in the NAS BT benchmark.

The sampling-based page migration mechanism is implemented via the use of a memory management thread that wakes up periodically and scans a fraction of the pages in the resident set of the program to detect pages candidate for migration. This mechanism makes no assumptions on the expected reference pattern of the program, and relies on the sampling frequency to adapt to the page reference pattern of the program as early as possible. The length of the sampling interval is a tunable parameter. Due to the inherent overhead of page migrations, which is in the order of 1ms. per migration on contemporary systems, the sampling interval must be in the order of at least a few hundred milliseconds. The amount of pages scanned upon each invocation of the memory manager thread is set by the runtime system to limit the cost of page migrations up to a small fraction (e.g. 10%) of the sampling interval.

In general, although fine-grain sampling intervals are expected to start migrating pages earlier to reach a good page distribution, a rather coarse-grain interval is expected to provide more robust behaviour, because it is likely to be less prone to transitory effects during execution, such as cold start, or a phase change. In the implementation we use by default a sampling interval of 1 second and the page migration engine scans 100 pages per invocation to locate pages candidate for migration.

These parameters work well in programs with execution times of at least a few minutes, since they amortize well the cost of migrations and enable the page migration engine to scan the entire address space in a small fraction of the total execution time. For programs with execution times in the order of a few seconds, shorter sampling intervals are required, however, for programs with large resident set sizes and short execution times, the minimum sampling interval of our implementation (100ms.) is not sufficient to optimize page placement in time and provide sizeable performance gains.

3.2 Data Redistribution

Compiler-driven data redistribution is emulated in UPMLIB via a mechanism called *record/replay*, which handles iterative parallel codes with possible phase changes in the page reference pattern. In the record/replay mode of execution, the compiler instruments the OpenMP program to record the page reference counters before and after the execution of code fragments, in which the program is expected to experience a phase change in the page reference pattern. Such phase changes occur for example in the NAS BT and SP benchmarks, during the execution of the discrete equation solvers in the z-direction, because the initial page placement is tuned to have good locality for the solvers in the x- and y- directions [2]. These phase changes are typically handled with page redistribution in HPF. The recording mechanism is activate for the first iteration of the parallel code. UPMLIB approximates data redistribution by comparing the two recorded sets of counter values before and after the phase change, and identifying competitively the pages that should move in order to tune page placement before the phase change. Using this information, UPMLIB replays the page migrations before the phase change and undoes the replayed migrations after the end of the phase change in subsequent iterations of the program.

With the record/replay mechanism page migrations need to reside on the critical path of the program, to tune page placement across phase changes. To limit the cost of page migrations, the record/replay mechanism moves only the n *most critical* pages, in each iteration, where n is a tunable parameter set experimentally, to balance the overhead of page migrations with the earnings from migrating pages. The n most critical pages are determined as follows: the pages are sorted in descending order of the ratio $\frac{racc_{max}}{lacc}$, where $lacc$ is the number of local accesses from the home node of the page and $racc_{max}$ is the maximum number of remote accesses from any of the other nodes. The pages that satisfy the inequality $\frac{racc_{max}}{lacc} > thr$, where thr is a predefined threshold are considered as eligible for migration. Let m be the number of these pages. If $n > m$, the n pages with the highest ratios $\frac{racc_{max}}{lacc}$ are migrated, else, the m candidate pages are migrated altogether. Figure 3 gives an example of how record/replay is used in the NAS BT benchmark for approximating the desired data redistribution before the execution of the `z_solve` function.

Likewise to the iterative page migration scheme described in Section 3.1, the record/replay mechanism is compliant to iterative parallel programs with repetitive reference patterns. Non-iterative programs can utilize sampling of page reference counters to respond to phase changes. However, in this case the sampling mechanism has two shortcomings. The first problem is that the appropriate sampling frequency depends on the granularity of the phase change in terms of execution time. For phase changes that last for at least a few hundreds of milliseconds, the sampling

```
...
call upmlib_init()
call upmlib_memrefcnt(u, size)
call upmlib_memrefcnt(rhs,size)
call upmlib_memrefcnt(forcing,size)
...
do step=1,niter
    call compute_rhs
    call x_solve
    call y_solve
    if (step .eq. 1) then
      call upmlib_record()
    else
      call upmlib_replay()
    endif
    call z_solve
    if (step .eq. 1) then
      call upmlib_record()
    else
      call upmlib_undo()
    endif
    call add
enddo
```

Figure 3: An example of the usage of the UPMlib record/replay mechanism in the NAS BT benchmark.

mechanism is likely to have enough time to migrate some pages in the direction of improving data locality. For finer-grain phase changes, the sampling mechanism is unlikely to move pages in time for improving performance. We note that this would also be the case for any data redistribution engine, since this engine should use the same OS mechanisms for moving data between nodes. The second problem is that as soon as a phase change occurs in the program, the page reference counters become obsolete, i.e. the reference history reflected by the counters is biased by the reference pattern of the program before the phase change. We circumvent this problem in *UPMlib* by aging the reference counters before and after phase changes in order to discard obsolete reference history. Aging is implemented by simply subtracting the values of the counters obtained before the phase change from the values obtained after the phase change.

3.3 Hybrid Schemes

In analogy to data distribution and redistribution, the mechanisms described in Sections 3.1 and 3.2 respectively can be combined to obtain the best of the two functionalities in OpenMP programs.

Figure 4 illustrates a possible usage of both the iterative and the record/replay mechanism for coarse-grain and fine-grain optimization of the NAS BT benchmark. In this example, the iterative page migration mechanism is used in the first few iterations of the benchmark, to optimize page placement with respect to the complete parallel computation. The record/replay mechanism is ac-

```
...
call upmlib_init()
call upmlib_memrefcnt(u, size)
call upmlib_memrefcnt(rhs,size)
call upmlib_memrefcnt(forcing,size)
...
do step=1,niter
    call compute_rhs
    call x_solve
    call y_solve
    if (step .lt. 10) then
      call upmlib_migrate_memory()
    else if (step .eq. 10) then
      call upmlib_record()
    else
      call upmlib_replay()
    endif
    call z_solve
    if (step .eq. 10) then
      call upmlib_record()
    else if (step .gt. 10) then
      call upmlib_undo()
    endif
    call add
enddo
```

Figure 4: Combining data distribution and redistribution via user-level page migration in the NAS BT benchmark.

tivated afterwards to optimize across the phase change in the `z_solve` function, assuming that the iterative mechanism has already reached a stable page distribution. Other schemes can also be applied. For example, in the case of non-iterative parallel programs, the page migration engine could use multiple sampling intervals, e.g. a long sampling interval for global optimization of page placement and a short sampling interval for local optimization of page placement between phase changes.

4 Experimental Results

We provide a set of experimental results that substantiate our arguments for dynamic page migration as a substitute for page distribution and redistribution in OpenMP. Our results are constrained by the fact that we were able to experiment only with iterative parallel codes —the OpenMP implementations of the NAS benchmarks as provided by their vendors [5]. Therefore, we follow a synthetic experimental approach for the cases in which the characteristics of the benchmarks do not meet the analysis requirements. All the experiments were conducted on 16 idle processors of a 64-processor SGI Origin2000 with 8 Gbytes of memory.

4.1 Data Distribution

We conducted the following experiment to assess the effectiveness of user-level dynamic page migration as a data distribution engine. We used five optimized OpenMP implementation of the NAS BT, SP, CG, MG and FT, which were customized to exploit the first-touch page placement scheme of the SGI Origin2000 [7]. Considering first-touch as the page placement scheme that achieves the best data distribution for these codes, we ran the codes using three alternate page placement schemes, namely round-robin page placement, random page placement and worst-case page placement.

First-touch and round-robin were available as runtime options, set with the `_DSM_PLACEMENT` environment variable of the Cellular IRIX operating system. To emulate random page placement, we invalidate the pages of all the shared arrays, by calling `mprotect()` with the `PROT_NONE` parameter. We install a `SIGSEGV` signal handler, which upon receiving a segmentation violation signal for a page, maps the page at a randomly selected node in the system. For benchmarks with resident set size in the order of a few thousand pages[1], a simple random generator is sufficient to produce a fairly balanced random distribution of pages among nodes.

The worst-case page placement is emulated by enabling first-touch page placement and forcing the first iteration of the complete computation in the codes, to run on one processor. With this trick, all the pages of the shared arrays are mapped on a single node of the system and subsequently accessed solely with remote accesses by all the processors, but the ones on the node on which the pages are mapped. Placing all pages on one node maximizes the rate of remote memory accesses and exacerbates contention at the memory module in which the pages reside.

Figure 5 shows the results from executing the OpenMP implementations of the NAS benchmarks with four page placement schemes. Each bar is an average of three independent experiments. The variance in all cases was negligible. The black bars illustrate the execution time with the different page placement schemes labeled as `ft-`, `rr-`, `rand-` and `sn-`, for first-touch, round-robin, random, and worst-case (single-node) page placement respectively. The light gray bars illustrate the execution time with the same page placement scheme and the IRIX page migration engine enabled during the execution of the benchmarks. The dark gray bars illustrate the execution time with the UPMLIB iterative page migration mechanism enabled in the benchmarks. The straight line in each chart, shows the baseline performance with the native first-touch page placement scheme of IRIX.

The results demonstrate that using a page placement scheme other than first-touch has modest to significant impact on performance. Worst-case page placement incurs a significant slowdown for all benchmarks except BT, in which the slowdown is modest. The average slowdown with worst-case page placement is 90%. Round-robin and random page placement have modest impact on performance, incurring slowdowns between 2% and 45%. The critical observation is the effectiveness of the two page migration engines, in closing the performance gap between first-touch and the other three page placement schemes. The IRIX page migration engine improves the performance of the suboptimal page placement schemes, but is unable to approach the performance of the best page placement and still incurs sizeable slowdowns of, on average 16%, 17% and 61% for round-robin, random and worst-case page placement respectively.

[1] We used the Class A problem size of the NAS benchmarks for experimentation.

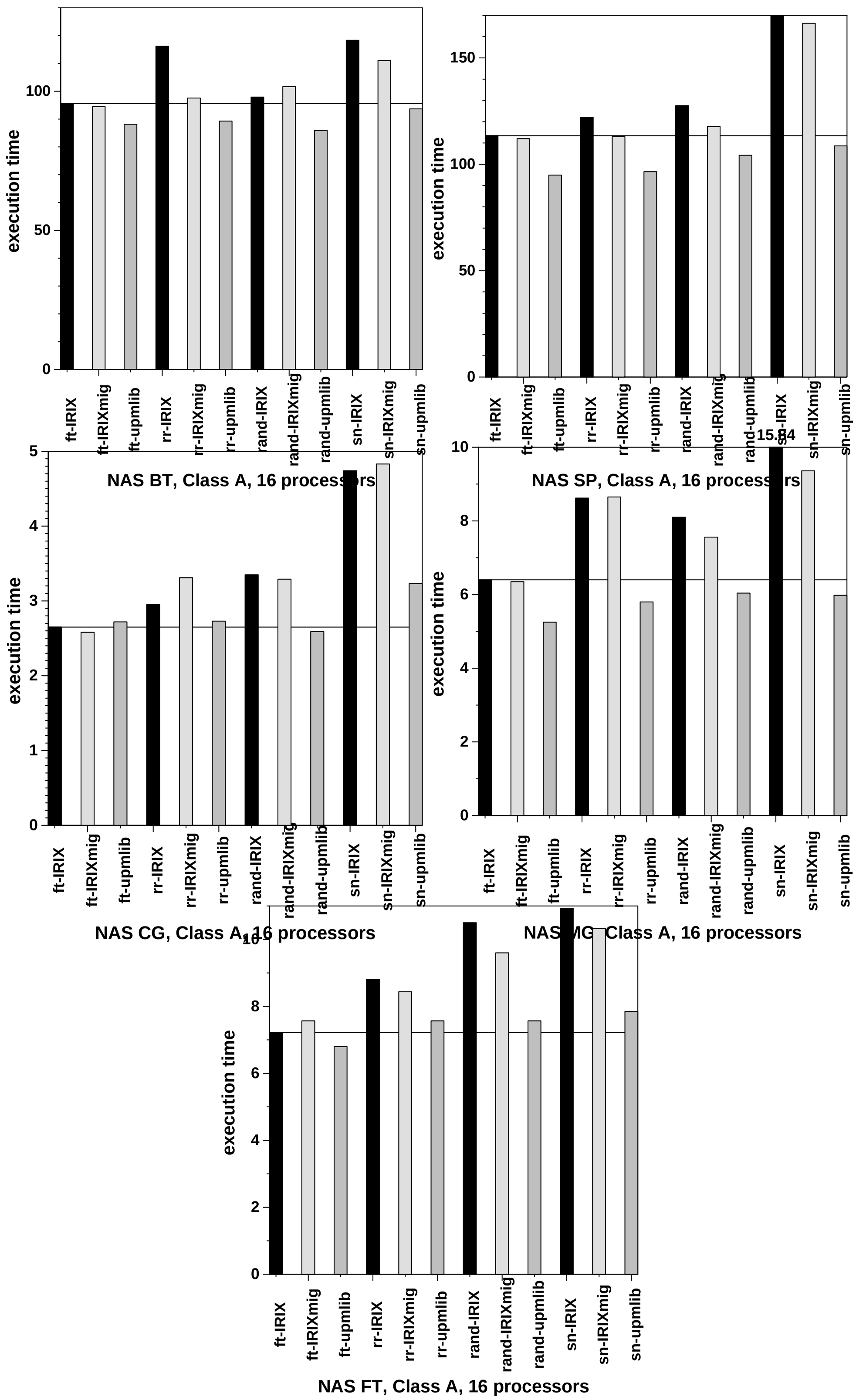

Figure 5: Performance of UPMlib with different page placement schemes.

On the other hand, the slowdown incurred from *UPMlib* compared to first-touch is much lower compared to the slowdown incurred with the IRIX page migration engine. When the codes are linked with UPMLIB, the programs are slowed down on average by 5% with round-robin page placement, 6% with random page placement and 18% with worst-case page placement. OpenMP programs become practically immune to the page placement strategy when the user-level dynamic page migration engine is enabled. This implies that with a smart dynamic page migration engine, it is possible to achieve performance close to the one obtained with the best data distribution scheme, without modifying the OpenMP standard and compromising its simplicity.

In order to assess the effectiveness of the sampling-based page migration engine of *UPMlib*, we conducted the following experiment. We activate the sampling mechanism for the NAS benchmark and compared the performance obtained with the sampling mechanism, against the performance obtained with the iterative page migration mechanism. The iterative mechanism is tuned to exploit the computational structure of the NAS benchmarks and can therefore serve as a meaningful performance boundary for the sampling mechanism.

In the experiments with the sampling mechanism, we used different sampling frequencies, according to the execution time of each benchmark. For BT and SP, the runtime of which are approximately one and a half minute, we used a sampling frequency of 1 second. For CG, MG, and FT, the runtime of which is 2.5 to 8 seconds, we used a sampling frequency of 300 milliseconds. The sampling rate was set to 200 pages per second in the first case and 100 pages per second in the second. All the aforementioned parameters were set after experimentation.

Figure 6 illustrates the execution times obtained from the sampling mechanism, in comparison to the execution times obtained with the iterative mechanism. We observe two different trends. For the relatively long-running codes, namely BT and SP, the sampling mechanism obtains essentially identical performance to the iterative mechanism. However, for short-running codes, although we use a fine-grain sampling frequency, the sampling mechanism performs significantly worse than the iterative mechanism. This result indicates that the sampling mechanism is likely to be effective at a coarse-grain time scale, either for long-running programs, or for programs with coarse phase changes, but it seems to be inappropriate for fine-grain programs, since there is not enough time for the page migration engine to move the pages and optimize memory performance in time.

4.2 Data Redistribution

We evaluated the ability of our user-level page migration engine to substitute data redistribution, by activating the record/replay mechanism of *UMPlib* in the NAS BT and SP benchmarks, before and after the execution of the `z_solve` function, during which both programs exhibit a phase change in the access pattern. We set the number of critical pages (cf. Section 3.2) to $n = 20$.

Figure 7 illustrates the performance of the record/replay mechanism with first-touch page placement and our user-level iterative page migration mechanism enabled (labeled ft-recrep in the chart). The striped part of the bars shows the non-overlapped overhead of the page migrations performed by the record/replay mechanism. For illustrative purposes the figure shows also the execution time of BT and SP with first-touch and the IRIX page migration engine, as well as the execution time

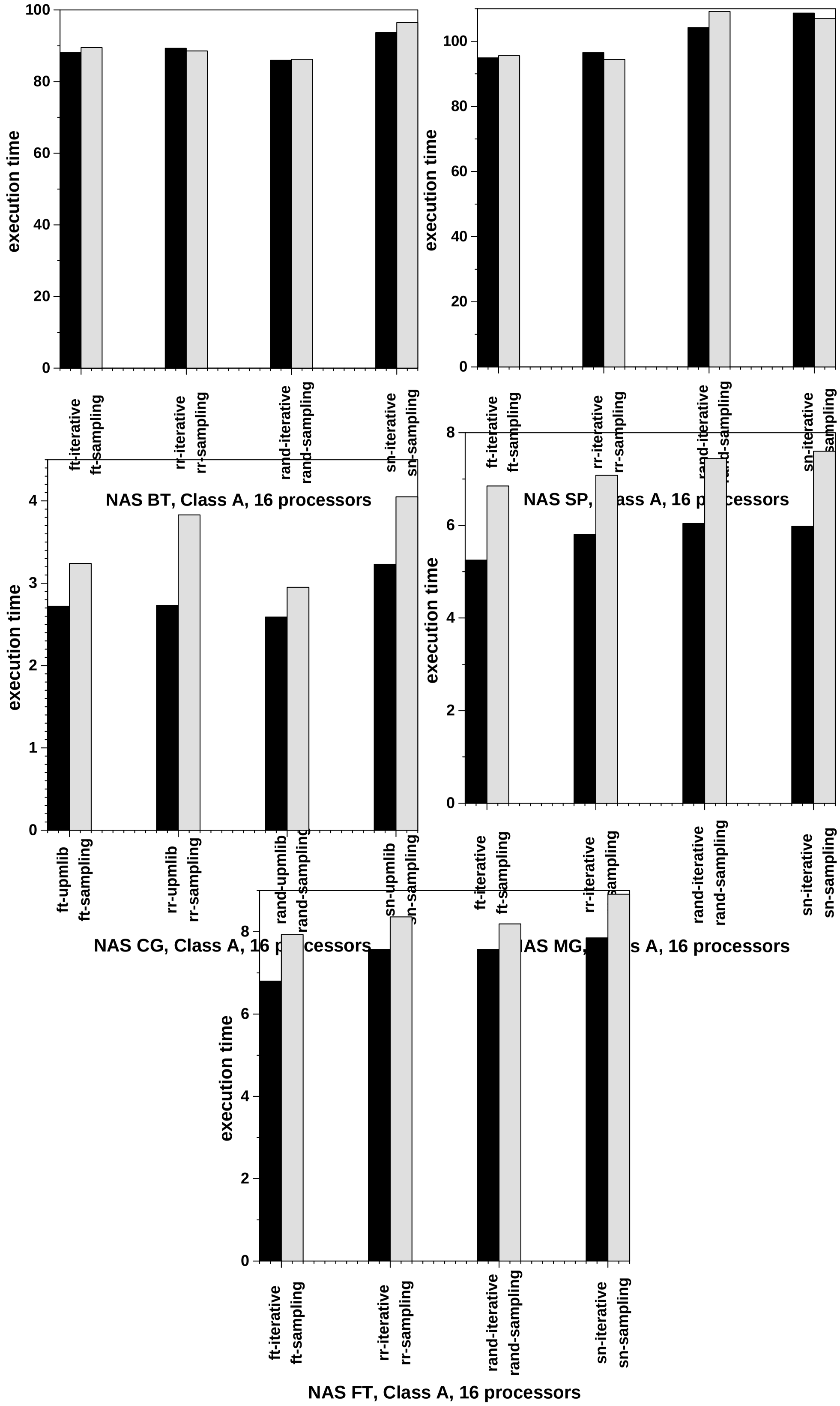

Figure 6: Performance of the sampling page migration mechanism of UPMlib with different page placement schemes.

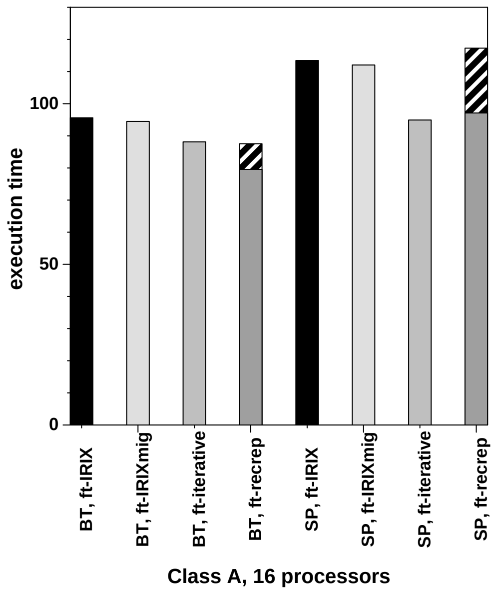

Figure 7: Performance of the record/replay mechanism for NAS BT and SP

.

with the iterative page migration mechanism of UPMLIB. The results indicate that applying page migration for fine-grain tuning of page placement across phase changes in the computation is non-profitable due to its excessive overhead. The overhead of the page migrations performed by the record/replay mechanism outweighs the gains from improving locality. Record/replay appears to suffer from the same problem like the sampling mechanism with fine-grain sampling intervals. A more detailed analysis of the codes, reveals that in both BT and SP the phase changes has a duration of approximately 300 ms. The record/replay mechanism has not enough time to optimize globally the page placement for the phase change. On the contrary it is forced to migrate a very small fraction of hot pages (only 20) and even in this case it suffers from the overhead. Note that we have experimented with larger values for n, and we observed serious performance deterioration in the benchmarks attributed to the page migration overhead. For example, setting $n = 50$ results to 20000 page migrations on the critical path of BT, which translates into a net overhead of 20 seconds.

4.3 Hybrid Schemes

In order to evaluate integrated schemes of implicit data distribution and redistribution with dynamic page migration, we instrumented NAS BT and SP to use the iterative page migration mechanism in the first few iterations of the code for optimizing the initial page placement and subsequently utilize the record/replay mechanism for tuning page placement before and after the phase change in the `z_solve` function, as illustrated in Figure 4.

Figure 8 illustrates the results.The charts repeat the results from Figure 7 and puts the performance of the hybrid scheme in comparison with the iterative and the record/replay schemes in isolation. The results show that although the hybrid scheme outperforms the record/replay scheme (marginally for BT and significantly for SP), it is still biased by the overhead of page migrations on the critical path of the programs, while replaying and undoing the recorded page migrations.

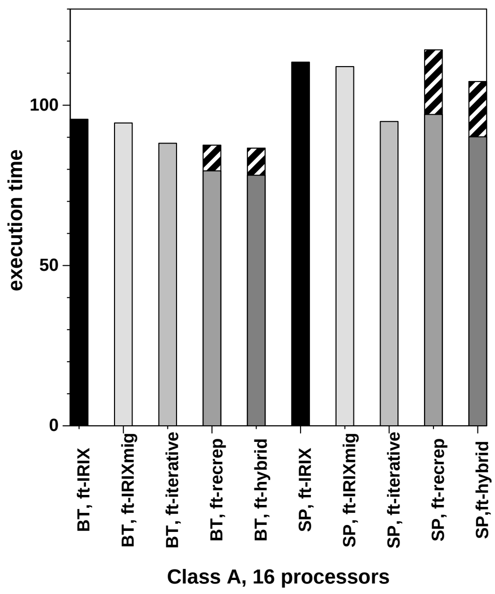

Figure 8: Performance of the hybrid page migration scheme in NAS BT and SP

.

5 Conclusion

This paper described and evaluated the mechanisms offered by UPMLIB for emulating data distribution and redistribution, features included in data parallel programming paradigms (for instance, High Performance Fortran). We have shown that the losses from poor initial page placement can be gained back with a smart user-level page migration scheme, without intervention from the programmer, or modification of the OpenMP standard. Our results demonstrate clearly that the need for introducing data distribution directives in OpenMP is obscure and may not warrant the implementation and standardization costs. On the other hand, we have shown that page migration is effective for coarse-grain optimization of data locality but suffers from excessive overhead when applied for tuning page placement at fine-grain time scales. It is therefore critical to estimate the cost/performance tradeoffs of page migration in order to investigate up to which extent can aggressive page migration strategies work profitably in place of data distribution and resdistribution on ccNUMA systems.

Acknowledgments

This work was supported by the European Commission, through the TMR Contract ERBFMGECT-950062 and in part by the ESPRIT IV Project No. 21907 (NANOS), the Greek Secretariat of Research and Technology through Project No. ED-99-566 and the Spanish Ministry of Education through Project No. TIC98-511. The experiments were conducted with resources provided by the European Center for Parallelism of Barcelona (CEPBA).

References

[1] R. Chandra et.al. Data Distribution Support for Distributed Shared Memory Multiprocessors. *Proc. of the 1997 ACM Conference on Programming Languages Design and Implementation*, pp. 334–345, Las Vegas, NV, June 1997.

[2] M. Frumkin, H. Jin and J. Yan. *Implementation of the NAS Parallel Benchmarks in High Performance FORTRAN*. Technical Report NAS-98-009, NASA Ames Research Center, September 1998.

[3] Y. Hu, H. Lu, A. Cox and W. Zwaenepoel. OpenMP on Networks of SMPs. *Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pp. 302–310, San Juan, Puerto Rico, April 1999.

[4] D. Jiang and J. P. Singh. Scaling Application Performance on a Cache-Coherent Multiprocessor. *Proc. of the 26th International Symposium on Computer Architecture*, pp. 305–316, Atlanta (USA), May 1999.

[5] H. Jin, M. Frumkin and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Technical Report NAS-99-011, NASA Ames Research Center, 1999.

[6] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr, and M. Zosel. The High Performance Fortran Handbook. The MIT Press, Cambridge, MA, 1994.

[7] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. *Proc. of the 24th International Symposium on Computer Architecture*, pp. 171–181, Denver(USA), May 1997.

[8] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta and E. Ayguadé. A Case for User-Level Dynamic Page Migration. *Proc. of the 14th ACM International Conference on Supercomputing*, Santa Fe, NM, May 2000.

[9] 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. *Proc. of the 5th ACM Workshop on Languages, Compilers and Runtime Systems for Scalable Computers*, Rochester, NY, May 2000.

[10] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta and E. Ayguadé. User-Level Dynamic Page Migration for Multiprogrammed Shared-Memory Multiprocessors. *Submitted for Publication*, January 2000.

[11] OpenMP Architecture Review Board. *OpenMP Specifications*. `http://www.openmp.org`. Accessed April 2000.

[12] Silicon Graphics Inc. Technical Publications. IRIX 6.5.5 man pages. `proc(4), mmci(5), schedctl(2).http://techpubs.sgi.com`, Accessed November 1999.

[13] B. Verghese, S. Devine, A. Gupta and M. Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. *Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 279–289, Cambridge(USA), October 1996.