

Leveraging Transparent Data Distribution in OpenMP via User-Level Dynamic Page Migration^{*}

Dimitrios S. Nikolopoulos¹, Theodore S. Papatheodorou¹,
Constantine D. Polychronopoulos², Jesús Labarta³, and Eduard Ayguadé³

¹ Department of Computer Engineering and Informatics
University of Patras, Greece
{dsn,tsp}@hpclab.ceid.upatras.gr

² Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
cdp@csrd.uiuc.edu

³ Department of Computer Architecture
Technical University of Catalonia, Spain
{jesus,eduard}@ac.upc.es

Abstract. This paper describes transparent mechanisms for emulating some of the data distribution facilities offered by traditional data-parallel programming models, such as High Performance Fortran, in OpenMP. The vehicle for implementing these facilities in OpenMP without modifying the programming model or exporting data distribution details to the programmer is user-level dynamic page migration [9,10]. We have implemented a runtime system called *UPMlib*, which allows the compiler to inject into the application a smart user-level page migration engine. The page migration engine improves transparently the locality of memory references at the page level on behalf of the application. This engine can accurately and timely establish effective initial page placement schemes for OpenMP programs. Furthermore, it incorporates mechanisms for tuning page placement across phase changes in the application communication pattern. The effectiveness of page migration in these cases depends heavily on the overhead of page movements, the duration of phases in the application code and architectural characteristics. In general, dynamic page migration between phases is effective if the duration of a phase is long enough to amortize the cost of page movements.

^{*} This work was supported by the E.C. through the TMR Contract No. ERBFMGECT-950062 and in part through the IV Framework (ESPRIT Programme, Project No. 21907, NANOS), the Greek Secretariat of Research and Technology (Contract No. E.D.-99-566) and the Spanish Ministry of Education through projects No. TIC98-511 and TIC97-1445CE. The experiments were conducted with resources provided by the European Center for Parallelism of Barcelona (CEPBA).

1 Introduction

One of the most important problems that programming models based on the shared-memory communication abstraction are facing on distributed shared-memory multiprocessors is poor data locality [3,4]. The non-uniform memory access latency of scalable shared-memory multiprocessors necessitates the alignment of threads and data of a parallel program, so that the rate of remote memory accesses is minimized. Plain shared-memory programming models hide the details of data distribution from the programmer and rely on the operating system for laying out the data in a locality-aware manner. Although this approach contributes to the simplicity of the programming model, it also jeopardizes performance, if the page placement strategy employed by the operating system does not match the memory reference pattern of the application. Increasing the rate of remote memory accesses implies an increase of memory latency by a factor of three to five and may easily become the main bottleneck towards performance scaling.

OpenMP has become the de-facto standard for programming shared-memory multiprocessors and is already widely adopted in the industry and the academia as a simple and portable parallel programming interface [11]. Unfortunately, in several case studies with industrial codes OpenMP has exhibited performance inferior to that of message-passing and data parallel paradigms such as MPI and HPF, primarily due to the inability of the programming model to control data distribution [1,12]. OpenMP provides no means to the programmer for distributing data among processors. Although automatic page placement schemes at the operating system level, such as first-touch and round-robin, are often sufficient for achieving acceptable data locality, explicit placement of data is frequently needed to sustain efficiency on large-scale systems [4].

The natural means to surmount the problem of data placement on distributed shared-memory multiprocessors is data distribution directives [2]. Indeed, vendors of scalable shared-memory systems are already providing the programmers with platform-specific data distribution facilities and the introduction of such facilities in the OpenMP programming interface is proposed by several vendors. Offering data distribution directives similar to the ones offered by High-performance Fortran (HPF) [7] in shared-memory programming models has two fundamental shortcomings. First, data distribution directives are inherently platform-dependent and thus hard to standardize and incorporate seamlessly in shared-memory programming models like OpenMP. OpenMP seeks for portable parallel programming across a wide range of architectures. Second, data distribution is subtle for programmers and compromises the simplicity of OpenMP. The OpenMP programming model is designed to enable straightforward parallelization of sequential codes, without exporting architectural details to the programmer. Data distribution contradicts this design goal.

Dynamic page migration [14] is an operating system mechanism for tuning page placement on distributed shared memory multiprocessors, based on the observed memory reference traces of each program at runtime. The operating system uses per-node, per-page hardware counters, to identify the node of the

system that references more frequently each page in memory. In case this node is other than the node that hosts the page, the operating system applies a competitive criterion and migrates the page to the most-frequently referencing node, if the page migration does not violate a set of resource management constraints. Although dynamic page migration was proposed merely as an optimization for parallel programs with dynamically changing memory reference patterns, it was recently shown that a smart page migration engine can also be used as a means for achieving good data placement in OpenMP without exporting architectural details to the programmer [9,10].

In this paper we present an integrated compiler/runtime/OS page migration framework, which emulates data distribution and redistribution in OpenMP without modifying the OpenMP application programming interface. The key for leveraging page migration as a data placement engine is the integration of the compiler in the page migration mechanism. The compiler can provide useful information on three critical factors that determine data locality: the areas of the address space of the program which are likely to concentrate remote memory accesses, the structure of the program, and the phase changes in the memory reference pattern. This information can be exploited to trigger a page migration mechanism at the points of execution at which data distribution or redistribution would be theoretically needed to reach good levels of data locality.

We show that simple mechanisms for page migration can be more than sufficient for achieving the same level of performance that an optimal initial data distribution scheme achieves. Furthermore, we show that dynamic page migration can be used for phase-driven optimization of data placement under the constraint that the computational granularity of phases is coarse enough to enable the migration engine to balance the high cost of coherent page movements with the earnings from reducing the number of remote memory accesses. The presented mechanisms are implemented entirely at user-level in *UPMlib* (User-level Page Migration library), a runtime system designed to tune transparently the memory performance of OpenMP programs on the SGI Origin2000. For details on the implementation and the page migration algorithms of *UPMlib* the reader is referred to [9,10]. This paper emphasizes the mechanisms implemented in *UPMlib* to emulate data distribution.

The rest of this paper is organized as follows. Section 2 shows how user-level page migration can be used to emulate data distribution and redistribution in OpenMP. Section 3 provides a set of experimental results that substantiate the argument that dynamic page migration can serve as an effective substitute for page distribution and redistribution in OpenMP. Section 4 concludes the paper.

2 Implementing Transparent Data Distribution at User-Level

This section presents mechanisms for emulating data distribution and redistribution in OpenMP programs without programmer intervention, by leveraging dynamic page migration at user-level.

2.1 Initial Data Distribution

In order to approximate an effective initial data distribution scheme, a page migration engine must be able to identify early in the execution of the program the node in which each page should be placed according to the expected memory reference trace of the program. Our user-level 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 and repeat exactly the same computation for a number of iterations, typically corresponding to time steps. This class of programs represents the vast majority of parallel codes. The second mechanism is designed for non-iterative programs and programs which are iterative but do not repeat the same reference trace in every iteration. Both mechanisms operate on ranges of the virtual address space of the program which are identified as *hot* memory areas by the OpenMP compiler. In the current setting, hot areas are the shared arrays which are both read and written in possibly disjoint OpenMP `PARALLEL DO` and `PARALLEL SECTIONS` constructs.

The iterative mechanism is activated by having the OpenMP compiler instrument the programs to invoke the page migration engine at the end of every outer iteration of the computation. At these points, the page migration engine obtains an accurate snapshot of the complete page reference trace of the program after the execution of the first iteration. Since the recorded reference pattern will repeat itself throughout the lifetime of the program, the page migration engine can use it to place any given page in an optimal manner, 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-pong'ing between more than one nodes due to page-level false sharing. This problem can be solved easily in the first few iterations of the program by freezing the pages that tend to bounce between nodes [9].

The iterative mechanism makes very accurate page migration decisions and amortizes well the cost of page migrations, since all the page movement activity is concentrated in the first iteration of the parallel program. This is also the reason that this mechanism is an effective alternative to an initial data distribution scheme. *UPMlib* actually deactivates the mechanism after detecting that page placement is stabilized and no further page migrations are needed to reduce the rate of remote memory accesses. Figure 1 gives an example of the usage of the iterative page migration mechanism in the NAS BT benchmark. In this example, `u`, `rhs` and `forcing` are identified as hot memory areas by the compiler and monitoring of page references is activated on these areas via the `upmlib_memrefcnt()` call to the runtime system. The function `upmlib_migrate_memory()` applies a competitive page migration criterion on all pages in the hot memory areas and moves the pages that satisfy this criterion [9].

In cases in which the complete memory reference pattern of a program cannot be accurately identified, *UPMlib* uses a mechanism which samples periodically the memory reference counters of a number of pages and migrates the pages that appear to concentrate excessive remote memory accesses. The sampling-based

```

...
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

```

Fig. 1. Using the iterative page migration mechanism of *UPMlib* in NAS BT

page migration mechanism is implemented with a memory management thread that wakes up periodically and scans a fraction of the pages in the hot memory areas to detect pages candidate for migration. The length of the sampling interval and the amount of pages scanned upon each invocation are tunable parameters. Due to the cost of page migrations, the duration of the sampling interval must be at least a few hundred milliseconds, in order to provide the page migration engine with a reasonable time frame for migrating pages and moving the cost of some remote accesses off the critical path.

The effectiveness of the sampling mechanism depends heavily on the characteristics of the temporal locality of the program at the page level. The coarser the temporal locality, the better the effectiveness of the sampling mechanism. Assume that the cost of a page migration is 1 ms (typical value for state of the art systems) and a program has a resident set of 3000 pages (typical value for popular benchmarks like NAS). In a worst-case scenario in which all the pages are misplaced, the page migration engine needs 3 seconds to fix the page placement, if the memory access pattern of the program remains uniform while pages are being moved by the runtime system. Clearly, if the program has execution time or phases of duration less than 3 seconds, there is not enough time for the page migration engine to move the misplaced pages. The sampling mechanism is therefore expected to have robust behaviour for programs with reasonably long execution times or reasonably short resident sets with respect to the cost of coherent page migration by the operating system.

2.2 Data Redistribution

Data redistribution in data parallel programming models such as HPF requires identification of phase changes in the reference pattern of the programs. In analogy to HPF, a phase in OpenMP can be defined as a sequence of basic blocks in which the program has a uniform communication pattern among processors.

Each phase may encapsulate more than one OpenMP `PARALLEL` constructs. Under this simple definition the OpenMP compiler can use the page migration engine to establish implicitly an appropriate page placement scheme before the beginning of each phase. The hard problem that has to be addressed in this case is how can the page migration engine identify a good page placement for each phase in the program, using only implicit memory reference information available from the hardware counters.

UPMlib uses a mechanism called *record/replay* to address the aforementioned problem. This mechanism is conceptually similar to the record/replay barriers described in [6]. The record/replay mechanism handles effectively strictly iterative parallel codes in which the same memory reference trace is repeated for a number of iterations. For non-iterative programs, or iterative programs with non-repetitive access patterns, *UPMlib* employs the sampling mechanism outlined in Section 2.1. The record/replay mechanism is activated as follows. The compiler instruments the OpenMP program to record the page reference counters at all phase transition points. The recording procedure stores two sets of reference traces per phase, one at the beginning of the phase and one before the transition to the next phase. The recording mechanism is activated only during the first iteration. *UPMlib* estimates the memory reference trace of each phase by comparing the two sets of counters that were recorded at the phase boundaries. The runtime system identifies the pages that should move in order to tune page placement before the transition to a phase, by applying the competitive criterion to all pages accessed during the phase, based on the corresponding reference trace. After the last phase of the first iteration, the program can simply undo the page migrations executed at all phase transition points by sending the pages back to their original homes. This action recovers the initial page placement scheme. In subsequent iterations, the runtime system replays the recorded page migrations at the respective phase transition points.

Figure 2 gives an example of how the record/replay mechanism is used in the NAS BT benchmark. BT has a phase change in the routine `z_solve`, due to the alignment of data in memory, which is done along the `x` and `y` dimensions. The page reference counters are recorded before and after the first execution of `z_solve` and the recorded values are used to identify page migrations which are replayed before every execution of `z_solve` in subsequent iterations. The routine `upmlib_undo()` is used to undo the page migrations performed by `upmlib_replay()`, in order to recover the initial page placement scheme that is tuned for `x_solve`, `y_solve` and `add`.

With the record/replay mechanism, page migrations necessarily reside on the critical path of the program. The mechanism is sensitive to the granularity of phases and is expected to work in cases in which the duration of each phase is long enough to amortize the cost of page migrations. In order to limit this cost, the record/replay mechanism can optionally move 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 reducing the rate of remote memory accesses. The *n* most critical pages are determined as follows: the pages

```

...
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

```

Fig. 2. Using the UPMLib record/replay mechanism in NAS BT

are sorted in descending order according to the ratio $\frac{r_{acc_{max}}}{l_{acc}}$, where l_{acc} is the number of local accesses from the home node of the page and $r_{acc_{max}}$ is the maximum number of remote accesses from any of the other nodes. The pages that satisfy the inequality $\frac{r_{acc_{max}}}{l_{acc}} > thr$, where thr is a predefined threshold are considered as eligible for migration. Let m be the number of these pages. If $m > n$, the mechanism migrates the n pages with the highest ratios $\frac{r_{acc_{max}}}{l_{acc}}$. Otherwise, the mechanism migrates the m eligible pages.

Similarly to data distribution and redistribution, the mechanisms described in Sections 2.1 and 2.2 can be combined effectively to obtain the best of the two functionalities in OpenMP programs. For example, the iterative page migration mechanism can be used in the first few iterations of a program to establish quickly a good initial page placement. The record/replay mechanism can be activated afterwards to optimize page placement across phase changes.

3 Experimental Results

We provide a set of experimental results that substantiate our argument that dynamic page migration is an effective 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 developers [5].

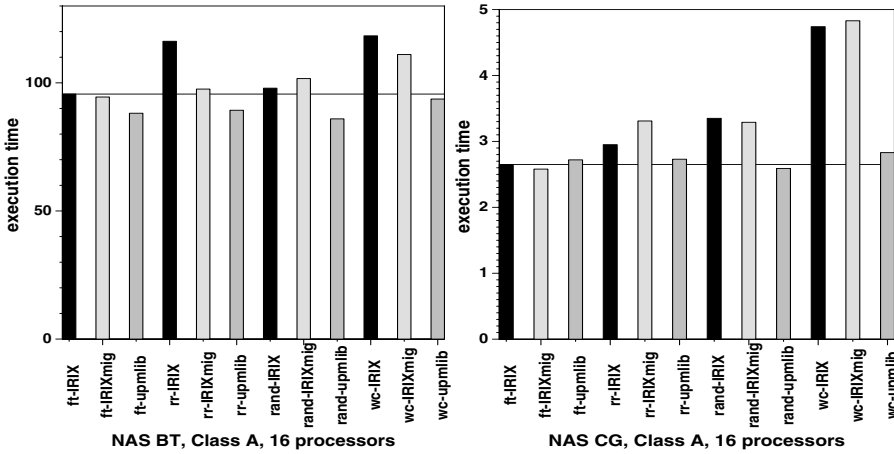


Fig. 3. Performance of UPMLib with different page placement schemes

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 MIPS R10000 processors running at 250 MHz and 8 Gbytes of memory. The system ran version 6.5.5 of the SGI IRIX OS.

3.1 Data Distribution

We conducted the following experiment to assess the effectiveness of the iterative mechanism of *UPMLib*. We used the optimized OpenMP implementations of five NAS benchmarks (BT, SP, CG, MG, FT), which were customized to exploit the first-touch page placement scheme of the SGI Origin2000 [8]. Considering first-touch as the page placement scheme that achieves the best data distribution for these codes, we ran the codes using three alternative page placement schemes, namely round-robin page placement, random page placement and worst-case page placement. Round-robin page placement could be optionally requested via an environment variable. Random page placement was hand-coded in the benchmarks, using the standard UNIX page protection mechanism to capture page faults and relocate pages, thus bypassing the default operating system strategy. Worst-case page placement was forced by a sequential execution of the cold-start iteration of each program, during which all data pages were placed on a single node of the system.

Figure 3 shows the results from executing the OpenMP implementations of the NAS BT and CG benchmarks with four page placement schemes. The observed trends are similar for all the NAS benchmarks used in the experiments. We omit the charts for the rest of the benchmarks due to space considerations. 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 dif-

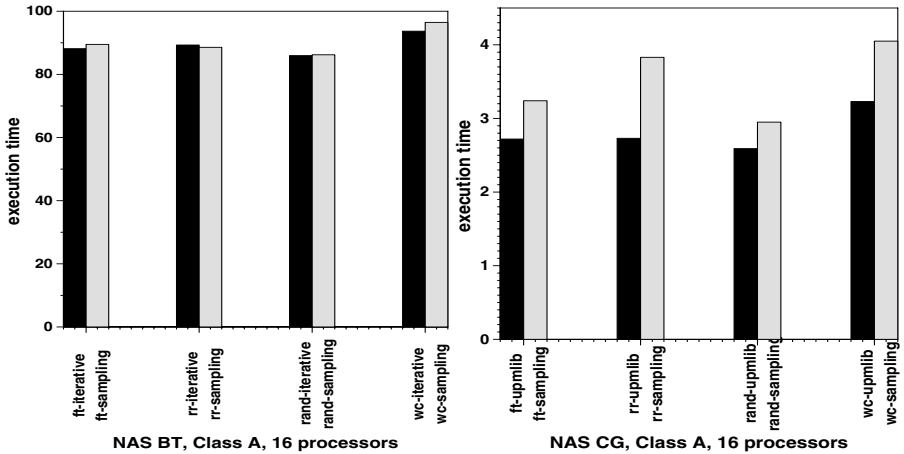


Fig. 4. Performance of the sampling page migration mechanism of UPMLib

ferent page placement schemes, labeled as *ft-IRIX*, *rr-IRIX*, *rand-IRIX* and *wc-IRIX*, for first-touch, round-robin, random, and worst-case 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 (same labels with suffix *-IRIXmig*). The dark gray bars illustrate the execution time with the *UPMLib* iterative page migration mechanism enabled in the benchmarks (same labels with suffix *-upmlib*). The horizontal lines show the baseline performance with the native first-touch page placement scheme of IRIX.

The results show that page placement schemes other than first-touch incur significant slowdowns compared to first-touch, ranging from 24% to 210%. The same phenomenon is observed even when page migration is enabled in the IRIX kernel, although page migration generally improves performance. On the other hand, when the suboptimal page placement schemes are combined with the iterative page migration mechanism of *UPMLib*, they approximate closely the performance of first-touch. When the page migration engine of *UPMLib* is injected in the benchmarks, the average performance difference between first-touch and the other page placement schemes is as low as 5%. In the last half iterations of the programs the performance difference was measured less than 1%. This practically means that the iterative page migration mechanism approaches rapidly the best initial page placement in each program. It also means that the performance of OpenMP programs can be immune to the page placement strategy of the operating system as soon as a page migration engine can relocate early poorly placed pages. No programmer intervention is required to achieve this level of optimization.

In order to assess the effectiveness of the sampling-based page migration engine of *UPMLib*, we conducted the following experiment. We activated the sampling mechanism in the NAS benchmarks and compared the performance

obtained with the sampling mechanism against the performance obtained with the iterative mechanism. The iterative mechanism is tuned to exploit the structure of the NAS benchmarks and can therefore serve as a meaningful performance boundary for the sampling mechanism.

Figure 4 illustrates the execution times obtained with the sampling mechanism, compared to the execution times obtained with the iterative mechanism in the NAS BT and CG benchmarks. BT is a relatively long running code with execution time in the order of one and a half minute. On the other hand, CG's execution time is only a few seconds. For BT, we used a sampling frequency of 100 pages per second. For CG, we used a sampling frequency of 100 pages per 300 milliseconds. In the case of BT, the sampling mechanism is able to obtain performance essentially identical to that of the iterative mechanism. A similar trend was observed for SP, the execution time of which is similar to that of BT. However, for the short-running CG benchmark, despite the use of a higher sampling frequency the sampling mechanism performs consistently significantly worse than the iterative mechanism. The same happens with MG and FT. The results mainly demonstrate the sensitivity of the sampling mechanism to the execution characteristics of the applications. The sampling mechanism is unlikely to benefit short running codes.

3.2 Data Redistribution

We evaluated the ability of our user-level page migration engine to emulate data redistribution, by activating the record/replay mechanism of *UMPlib* in the NAS BT and SP benchmarks. Both benchmarks have a phase change in the execution of the `z_solve` function, as shown in Fig. 2. In these experiments, we restrict the page migration engine to move the n most critical pages across phases. The parameter n was set equal to 20.

Figure 5 illustrates the performance of the record/replay mechanism with first-touch page placement (labeled `ft-recrep` in the charts), as well as the performance of a hybrid scheme that uses the iterative page migration mechanism in the first few iterations of the programs and the record/replay mechanism in the rest of the iterations, as described in Section 2.2 (labeled `ft-hybrid` in the charts). The striped part of the bars labeled `ft-recrep` and `ft-hybrid` shows the non-overlapped overhead of 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 with the iterative page migration mechanism of *UPMlib*.

The results indicate that applying page migration for fine-grain tuning across phase changes may be non-profitable due to the excessive overhead of page movements in the operating system. In the cases of BT and SP the overhead of the record/replay mechanism appears to outweigh the gains from reducing the rate of remote memory accesses. A more detailed analysis of the codes reveals that the parallel execution of `z_solve` in BT and SP on 16 processors takes approximately 130 to 180 ms. The recording mechanism of *UPMlib* identifies between 160 and 250 pages to migrate before the execution of `z_solve`. The cost of a page

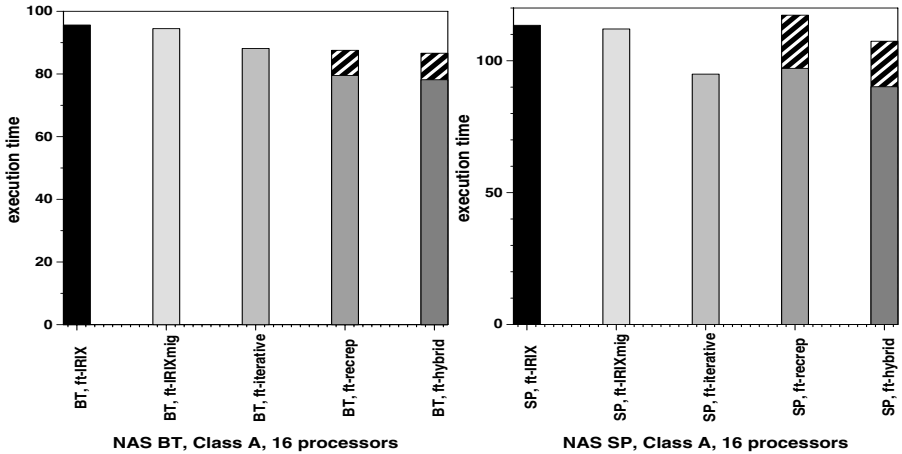


Fig. 5. Performance of the record/replay mechanism for NAS BT and SP

migration in the system we were experimenting with was measured to range between 1 and 1.3 ms, depending on the distance between the nodes that competed for the page. The total cost for moving all the pages identified by the recording mechanism as candidates for migration would exceed significantly the duration of the phase, making the record/replay mechanism useless.

Migrating the 20 most critical pages was able to reduce the execution time of useful computation by about 10% in the case of BT. However, the overhead of page migrations outweighed the earnings. The performance of the record/replay mechanism in the SP benchmark was disappointing. The limited improvements are partially attributed to the architectural characteristics of the Origin2000 and most notably the very low remote to local memory access latency ratio of the system, which is about 2:1 on the scale on which we experimented. The reduction of remote memory accesses would have a more significant performance impact on systems with higher remote to local memory access latency ratios. We have experimented with larger values for n and observed significant performance degradation, attributed again to the page migration overhead. The hybrid scheme appears to outperform the record/replay scheme (marginally for BT and significantly for SP), but it is still biased by the overhead of page migrations.

In order to quantify the extent to which the record/replay mechanism is applicable we executed the following synthetic experiment. We modified the code in NAS BT to quadruple the amount of work performed in each iteration of the parallel computation. We did not change the problem size of the program to preserve its locality characteristics. We rather enclosed each of the functions that comprise the main body of the computation (`x_solve`, `y_solve`, `z_solve`, `add`) in a loop. With this modification, we lengthened the duration of the parallel execution of `z_solve` to approximately 500 ms. Figure 6 shows the results from these experiments. It is evident that a better amortization of the overhead of page migrations helps the record/replay mechanism. In this experiment, the cost

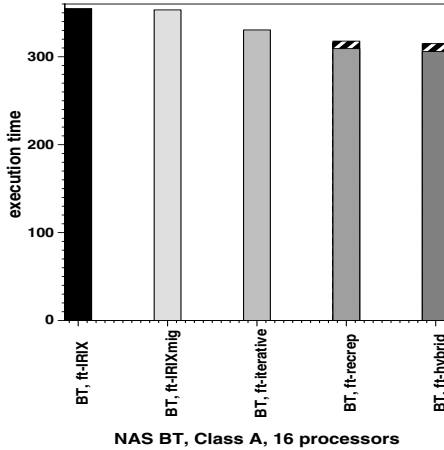


Fig. 6. Performance of the record/replay mechanism in the synthetic experiment with NAS BT

of the record/replay mechanism remains the same as in the previous experiments, however, the reduction of remote memory accesses achieved by the mechanism is exploited over a longer period of time. This yields a performance improvement of 5% over the iterative page migration mechanism.

4 Conclusion

This paper presented and evaluated mechanisms for transparent data distribution in OpenMP programs. The mechanisms leverage dynamic page migration as an oblivious to the programmer data distribution technique. We have shown that effective initial page placement can be established with a smart user-level page migration engine that exploits the iterative structure of parallel codes. 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 although page migration may be effective for coarse-grain optimization of data locality, it 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 effectively in place of data distribution and redistribution on distributed shared-memory multiprocessors. Since the same investigation would be necessary in data-parallel environments also, we do not consider it as a major restriction of our environment.

References

1. L. Brieger. *HPF to OpenMP on the Origin2000. A Case Study*. Proc. of the First European Workshop on OpenMP, pp. 19–20. Lund, Sweden, October 1999. 416
2. 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. 416
3. C. Holt, J. Pal Singh and J. Henessy. *Application and Architectural Bottlenecks in Large Scale Shared Memory Machines*. Proc. of the 23rd Int. Symposium on Computer Architecture, pp. 134–145. Philadelphia, PA, June 1996. 416
4. D. Jiang and J. Pal Singh. *Scaling Application Performance on a Cache-Coherent Multiprocessor*. Proc. of the 26th Int. Symposium on Computer Architecture, pp. 305–316. Atlanta, GA, May 1999. 416
5. H. Jin, M. Frumkin and J. Yan. *The OpenMP Implementation of NAS Parallel Benchmarks and its Performance*. Tech. Rep. NAS-99-011. NASA Ames Research Center, October 1999. 421
6. P. Keleher. *A High Level Abstraction of Shared Accesses*. ACM Transactions on Computer Systems, Vol. 18, No. 1, pp. 1–36. February 2000. 420
7. C. Koelbel et.al. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994. 416
8. J. Laudon and D. Lenoski. *The SGI Origin: A ccNUMA Highly Scalable Server*. Proc. of the 24th Int. Symposium on Computer Architecture, pp. 241–251. Denver, CO, June 1997. 422
9. D. Nikolopoulos et. al. *A Case for User-Level Dynamic Page Migration*. Proc. of the 14th ACM Int. Conference on Supercomputing, pp. 119–130. Santa Fe, NM, May 2000. 415, 417, 418
10. D. Nikolopoulos et. al. *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. 415, 417
11. OpenMP Architecture Review Board. *OpenMP Specifications*. <http://www.openmp.org>, accessed April 2000. 416
12. M. Resch and B. Sander. *A Comparison of OpenMP and MPI for the Parallel CFD Case*. Proc. of the First European Workshop on OpenMP. Lund, Sweden, October 1999. 416
13. Silicon Graphics Inc. Technical Publications. *IRIX 6.5 man pages*. `proc(4)`, `mmci(5)`, `schedct1(2)`. <http://techpubs.sgi.com>, accessed January 2000.
14. 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 Int. Conference on Architectural Support for Programming Languages and Operating Systems, pp. 279–289. Cambridge, MA, October 1996. 416