

# UPMLIB: A Runtime System for Tuning the Memory Performance of OpenMP Programs on Scalable Shared-Memory Multiprocessors\*

Dimitrios S. Nikolopoulos<sup>1</sup>, Theodore S. Papatheodorou<sup>1</sup>,  
Constantine D. Polychronopoulos<sup>2</sup>, Jesús Labarta<sup>3</sup>, and Eduard Ayguadé<sup>3</sup>

<sup>1</sup> Department of Computer Engineering and Informatics  
University of Patras, Greece

{dsn, tsp}@hpclab.ceid.upatras.gr

<sup>2</sup> Department of Electrical and Computer Engineering  
University of Illinois at Urbana-Champaign  
cdp@csr.d.uiuc.edu

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

**Abstract.** We present the design and implementation of UPMLIB, a runtime system that provides transparent facilities for dynamically tuning the memory performance of OpenMP programs on scalable shared-memory multiprocessors with hardware cache-coherence. UPMLIB integrates information from the compiler and the operating system, to implement algorithms that perform accurate and timely page migrations. The algorithms and the associated mechanisms correlate memory reference information with the semantics of parallel programs and scheduling events that break the association between threads and data for which threads have memory affinity at runtime. Our experimental evidence shows that UPMLIB makes OpenMP programs immune to the page placement strategy of the operating system, thus obviating the need for introducing data placement directives in OpenMP. Furthermore, UPMLIB provides solid improvements of throughput in multiprogrammed execution environments.

**Keywords:** OpenMP, scalable shared-memory multiprocessors, memory management, runtime systems, operating systems.

## 1 Introduction

Scalable shared-memory multiprocessor architectures converge remarkably to a common model, in which nodes with commodity microprocessors and memory are connected via a fast network and equipped with additional hardware support to provide

---

\* 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).

the communication abstraction of a shared address space to the programmer [2]. High-level programming models for scalable parallel computers converge also to a small set of standards that represent essentially two programming methodologies with different communication abstractions, namely message-passing and shared-memory. MPI [3] and OpenMP [13] are the most popular representatives of these programming methodologies.

There is a considerable debate going on recently with respect to the programming model of choice for scalable shared-memory multiprocessors. Interestingly, contemporary systems such as the SGI Origin2000 [8] support programming models based on both message-passing and shared-memory, via customized runtime systems provided by the vendors. Performance experiences on these systems indicate that implementations of parallel programs with MPI perform often better than implementations of the same programs with OpenMP. This is true especially for large industrial codes [14].

The most prominent problem that OpenMP are faced with on scalable shared-memory multiprocessors is the non-uniformity of memory access latencies (NUMA). Although the shared-memory communication abstraction hides data distribution details from the programmer, the programs are very sensitive to the page placement strategy of the operating system. A poor page placement scheme may exacerbate the number of remote memory accesses, which cost two to ten times as much as local memory accesses on state-of-the-art systems. It is therefore critical to ensure that threads and data are aligned in the nodes of the system, so that each thread is collocated with the data that the thread accesses more frequently.

Unfortunately, in order to achieve the aforementioned goal with a plain shared-memory programming model, the programmer must be aware of the page placement strategy of the operating system and either modify the program to adapt its memory reference pattern to the enforced system policy, or bypass the operating system and hand-code a customized page placement scheme [6]. Both approaches compromise the simplicity of shared-memory programming models and jeopardize their portability across different platforms. Nevertheless, vendors of shared-memory multiprocessors are already facing the dilemma of whether data distribution directives should be introduced in OpenMP or not [9].

The question that motivates the work presented in this paper is whether OpenMP can be enhanced with runtime capabilities for the transparent improvement of data locality at the page level, without exporting data distribution details to the programmer. We present the design and implementation of UPMLIB (User-Level Page Migration library), a runtime system with mechanisms and algorithms that transparently optimize at runtime the page placement of OpenMP programs, using feedback from the compiler, the operating system and dynamic monitoring of the memory reference pattern of the programs. UPMLIB leverages dynamic page migration [16] at user-level [10] to correct suboptimal page placement decisions made by the operating system.

The notable difference between UPMLIB and previously proposed kernel-level page migration engines, is that the employed dynamic page migration algorithms correlate the memory reference information obtained from hardware counters with the semantics of the parallel computation and scheduling information provided by the operating system. This is accomplished by integrating the compiler, the runtime system

and the operating system in the page migration engine. The compiler drives the page migration mechanism, by identifying memory regions which are likely to contain pages candidate for migration and instrumenting the programs to invoke the page migration engine of UPMLIB. The operating system provides scheduling notifications to the runtime system in order to trigger aggressive page migration schemes upon thread migrations. Thread migrations incur bursts of remote memory accesses due to cache reloads and the misalignment between migrated threads and the set of pages for which these threads exhibit memory affinity. The overall approach improves the accuracy and timeliness of page migrations, amortizes well the cost of page migrations over time, and makes the page migration engine responsive to unpredictable runtime events that may harm data locality. Furthermore, implementing the page migration engine entirely at user-level provides us with a great deal of flexibility in testing and customizing memory management schemes without requiring kernel source code and without compromising the well-tuned resource management policies of the operating system.

We have implemented UPMLIB on the SGI Origin2000, using the IRIX 6.5.5 memory management control interface. As a case study, we have used UPMLIB with unmodified OpenMP implementations of the NAS benchmarks [7]. Our results show that UPMLIB embeds the desirable immunity of OpenMP codes to the page placement strategies of the operating system. In addition, UPMLIB provides solid and in some cases significant performance improvements compared to the native IRIX page placement and migration schemes for standalone parallel programs and multiprogrammed workloads, scheduled with space- or time-sharing by the IRIX kernel.

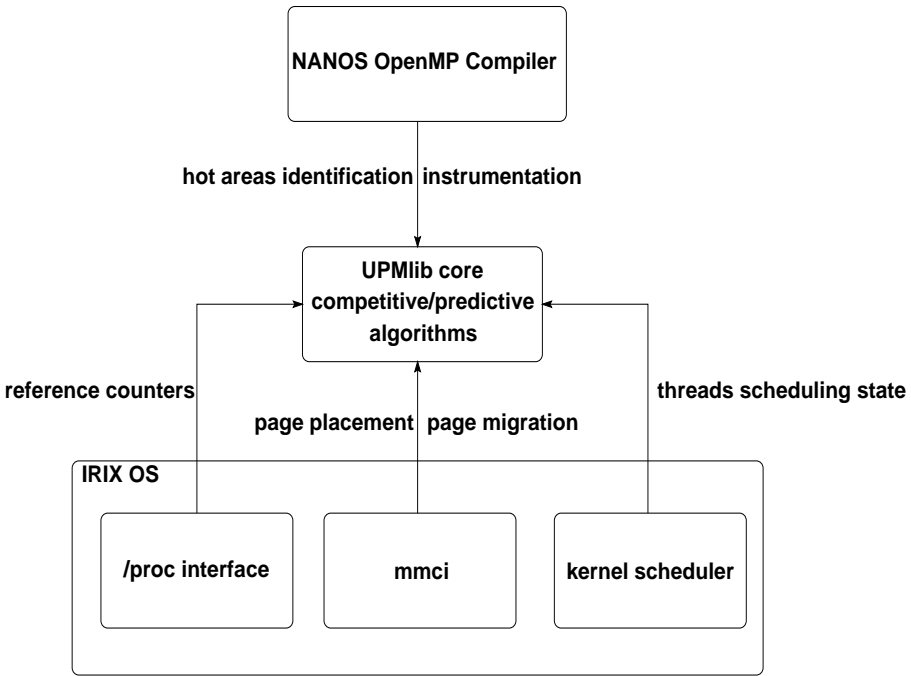
The rest of this paper is organized as follows. Section 2 outlines the design of UPMLIB. Section 3 provides implementation details. Section 4 presents results with OpenMP codes that utilize UPMLIB to improve their data locality in dedicated and multiprogrammed execution environments. Section 5 concludes the paper.

## 2 UPMLIB Design and Algorithms

The key design issue of UPMLIB is the integration of the compiler, the runtime system and the operating system in a unified framework that enhances the effectiveness of dynamic page migration. The page migration engine of UPMLIB correlates the dynamic reference pattern of a parallel program with the semantics of the program and the scheduling status of its threads at runtime. UPMLIB implements feedback-guided optimization of page placement in a local scope, in order to arm OpenMP programs with invulnerability to the global memory management strategy of the operating system and interventions of the kernel scheduler, when parallel programs are executed in multiprogrammed environments. Figure 1 shows the main modules and interfaces of UPMLIB. These are explained in detail in the following paragraphs.

### 2.1 Compiler Support

The OpenMP compiler identifies areas of the virtual address space which are likely to contain pages candidate for migration and instruments the programs to call the page



**Fig. 1.** UPMLIB modules and interfaces.

migration services of UPMLIB at specific points during their execution. In our first prototype, the compiler locates shared arrays which are both read and written in possibly disjoint sets of OpenMP parallel/work sharing constructs, thus incurring interprocessor communication of shared data. The compiler identifies these arrays as *hot* memory areas and inserts calls to UPMLIB for activating dynamic monitoring of page reference activity and page migration on these areas. The implementation is flexible enough to exploit advanced compiler knowledge, in case the compiler can provide accurate boundaries for parts of the hot areas which are likely to concentrate the most significant fraction of remote memory accesses, and the exact points of the program at which page migration could improve locality by emulating data distribution and redistribution schemes.

The compiler exploits the semantics of the parallel program in order to migrate pages accurately and ahead in time. The associated mechanisms distinguish between iterative and non-iterative parallel programs. The former represent the vast majority of parallel codes. For iterative programs, the compiler applies page migration at a coarse-grain scale, namely at the ends of the outer iterations of the parallel program. At these points of execution the runtime system can obtain an accurate view of the complete page reference pattern of the parallel computation by reading the hardware counters. Therefore, the runtime system is in a position to take successful decisions for migrating pages and achieve an *optimal* page placement, where *optimal* is defined with respect to the observed repetitive memory reference pattern of the program. The optimal page

placement is achieved when each page is placed in a node so that the maximum latency due to remote memory accesses by any node in the system to this page is minimized.

For strictly iterative parallel computations and in the absence of page-level false-sharing or thread migrations, the runtime system attains the best page placement with respect to the observed reference pattern after executing a single iteration of the parallel computation. Besides to the advantage of timeliness, this strategy amortizes well the cost of page migrations over time. Cost amortization is of particular importance, since page migrations are overly expensive operations on state-of-the-art systems. Since page migration is performed based on the reference trace of the complete parallel computation, the page migration engine is not biased by temporary effects such as cold-start or phase changes<sup>1</sup> in the reference pattern [10].

UPMLIB handles non-iterative codes, as well as iterative codes with non-repetitive access patterns, using a sampling-based mechanism for migrating pages. The runtime system wakes up periodically a thread, which scans a fraction of the pages in the hot memory areas and migrates some of these pages if needed. The sampling frequency and the amount of pages scanned upon each invocation of UPMLIB can be adjusted by the user to fit the characteristics of the application. Programs with frequent changes in the communication pattern between processors benefit from short sampling intervals, while programs with infrequent changes in the communication pattern can utilize longer sampling intervals. The amount of pages scanned upon each invocation is selected to limit the cost of checking and migrating pages to at most a small fraction of the sampling interval. The algorithm for scanning pages can vary from sequential to stride to randomized scanning, in order to enable the runtime system to adapt the page migration engine to the distribution of hot pages in the virtual address space.

Practically, the duration of the sampling interval must be at least a few hundred milliseconds. This holds due to the high cost of page migrations. The sampling interval must be selected to give the runtime system enough time to migrate a reasonable amount of pages ahead in time, so that a good fraction of the cost of remote memory accesses to these pages is moved off the critical path of the program. The sampling mechanism is beneficial for programs in which phases in the memory reference pattern last for at least a few seconds. Programs that exhibit fine-grain phase changes do not provide usually enough time to the runtime system for migrating pages.

The compiler is in a position to apply more aggressive data locality optimizations using page migration as its vehicle. As an example, the compiler can apply phase-driven optimization of page placement. The compiler can analyze the communication patterns of each phase, detect phase changes due to changes in the communication pattern across phases, and invoke the page migration mechanism between phases [5]. The effectiveness of such optimizations depends on the granularity of the phases in terms of execution time. The compiler analysis should be quite conservative when optimizing page placement across phase changes, because page migrations have to be performed on the critical path of the parallel program, thus making the amortization of the cost of the runtime system a critical performance parameter. Optimization of page placement across phase changes is a subject of investigation in our current version of UPMLIB.

---

<sup>1</sup> In OpenMP, we define a phase as a sequence of parallel or worksharing constructs that have the same communication pattern among processors.

## 2.2 Page Migration Algorithms

UPMLIB uses by default a competitive algorithm for migrating pages. The criterion of competitiveness in the algorithm is the estimated latency seen by each node in the system due to remote memory accesses. This criterion incorporates the number of references, the estimated cost of each remote reference according to the distance in hops between the referencing node and the referenced page, and contention at the nodes to which references are issued. The competitive thresholds used in the algorithm are tunable and may change at runtime, according to the observed effectiveness of page migrations on reducing the rate of remote memory accesses. In addition, the page migration algorithms include a self-deactivation mechanism, which disables the page migration mechanisms when it detects that the memory reference pattern is stabilized so that no further page migrations are needed by the runtime system. More details can be found in [10].

UPMLIB circumvents page-level false-sharing with a ping-pong prevention mechanism. The idea is to avoid migrating a page if it is likely to bounce between the same nodes more than once. The ping-pong prevention mechanism ensures that unless the threads of a parallel program migrate between nodes, each page will be placed at the appropriate node within the first two iterations of the program, assuming a strictly iterative program with a repetitive reference pattern. For the more general case in which pages can ping-pong between more than two nodes due to wide false-sharing, UPMLIB uses a bouncing threshold to limit the maximum number of times a page can move before the runtime system pins the page to a node. The bouncing threshold of UPMLIB is also a tunable parameter of the runtime system, to handle cases in which ping-pong of a page can actually be beneficial, for improving data locality across distinct phases.

## 2.3 Operating System Support

On scalable shared-memory multiprocessors, the page placement strategy establishes an implicit association between threads and data in a parallel program. In principle, a thread is associated with its *memory affinity set*, that is, the set of pages that the thread accesses more frequently than any other thread of the same program. On a multiprogrammed system in which multiple parallel and sequential programs execute simultaneously, the operating system arbitrarily preempts and migrates threads between nodes, thus breaking the association between these threads and their memory affinity sets. Thread migrations incur the cost of reloading the working sets of migrated threads from remote memory modules, as well as satisfying most cache misses incurred from migrated threads remotely. A page migration mechanism can alleviate this problem by forwarding the pages that belong to the memory affinity set of a migrated thread to the new node that hosts the thread. Unfortunately, a competitive page migration algorithm may fail to perform timely page migrations in this case. The reason is that the page reference counters may have accumulated obsolete reference history that prevents a page from migrating, unless the new home node of the migrated thread issues a sufficiently large amount of remote references to meet the competitive criterion.

UPMLIB uses a lightweight communication interface with the operating system to obtain scheduling information, which is used as a trigger for activating aggressive page

**Table 1.** UPMLIB interface.

Call	Functionality
<code>upmlib_init(), upmlib_end()</code>	UPMLIB initialization and termination.
<code>upmlib_memrefcnt(va, size)</code>	Initializes reference counting and activates dynamic page migration for the range <code>[va,va+size-1]</code> .
<code>upmlib_migrate_pages(pol)</code>	Runs the specified page migration policy for all hot memory areas.
<code>upmlib_check_pset()</code>	Polls the effective processor set on which the program executes from shared memory and records thread migrations.
<code>upmlib_switch()</code>	Switches the page migration policy from competitive to predictive and vice-versa using OS information.
<code>upmlib_record_counters()</code>	Records per-page/per-node reference counters for statistics collection.

forwarding algorithms upon migrations of threads from the operating system. The runtime system polls a vector in shared-memory which stores the instantaneous mapping of threads to processors and switches on the fly the default competitive algorithm, if it detects that some threads have migrated. In that case, UPMLIB activates a predictive algorithm which forwards pages in the memory affinity sets of migrated threads. The idea is to have the pages of a memory affinity set of a thread follow the thread in case this thread migrates. In the actual implementation, the runtime system detects *permanent* thread migrations, that is, thread migrations that move a thread to a node for an amount of time sufficiently long to justify the activation of the page forwarding mechanism. The associated algorithms and implementation issues are available in [11].

**Table 2.** UPMLIB environment variables.

Variable	Functionality
UMIGR_POLICY	Page migration criterion
UMIGR_THRESHOLD	Competitive criterion threshold
UMIGR_PING_PONG_LIMIT	Bouncing threshold for ping-pong
UMIGR_SAMPLING_PERIOD	Period for the sampling-based mechanism
UMIGR_PAGES_PER_SAMPLE	Number of pages sampled per invocation of the sampling-based mechanism
UMIGR_THREAD	Thread that executes UPMLIB code

Table 1 summarizes the UPMLIB user-level interface. This interface is meant to be used by the compiler, in the process of instrumenting OpenMP programs to use the page migration engine. Table 2 shows the runtime environment variables used to set the tunable parameters of UPMLIB. Figure 2 gives an example of the use of UPMLIB in the NAS BT benchmark. In this example, the compiler identifies three arrays of the application (`u`, `rhs`, `forcing`) as hot memory areas and activates the monitoring of page reference rates on these areas using the `upmlib_memrefcnt()` call to the runtime

```

call upmlib_init()
call upmlib_memrefcnt(u, size_of_u)
call upmlib_memrefcnt(rhs, size_of_rhs)
call upmlib_memrefcnt(forcing, size_of_forcing)
...
do step=1, niter
  call compute_rhs
  call x_solve
  call y_solve
  call z_solve
  call addi
  stat=upmlib_check_pset()
  if (stat .gt. 0) then
    call upmlib_switch(PREDICTIVE)
  else
    call upmlib_switch(COMPETITIVE)
  endif
  call upmlib_migrate_pages()
enddo

```

**Fig. 2.** Usage of UPMLIB in the NAS BT benchmark.

system. The function `upmlib_check_pset()` polls the scheduling information provided by the operating system and returns a positive value in case the operating system has performed at least one thread migration and the migrated thread has stayed on the same node for a sufficiently long amount of time. If no such thread migration has occurred, the default competitive page migration algorithm is invoked at the end of every iteration of the outer `do` loop, by calling the function `upmlib_migrate_pages()`. The page migration engine scans the hot memory areas, identifies pages candidate for migration and migrates pages according to the competitive criterion. In the event of a thread migration, the compiler switches the page migration algorithm to use the aggressive predictive criterion for page forwarding, by calling `upmlib_switch()`. The same function is called to switch back to the competitive algorithm in the absence of thread migrations by the operating system.

### 3 Implementation

UPMLIB is implemented on the SGI Origin2000, using the user-level memory management services of the Cellular IRIX operating system. The runtime system is integrated with the NANOS OpenMP compiler [1], which implements the instrumentation pass for using UPMLIB.

#### 3.1 Interfaces

The page migration facilities of UPMLIB use the memory management control interface (`mmci`) of IRIX (see Figure 1). The IRIX `mmci` provides significant flexibility

in managing physical memory at user-level, by virtualizing the topology of the system. The user can create high-level abstractions of the physical memory space, called *Memory Locality Domains* (MLDs). MLDs can be statically or dynamically mapped to physical nodes of the system. After establishing a mapping between MLDs and nodes, the user can associate ranges of the virtual address space with MLDs in order to implement application-specific page placement schemes. The runtime system requests the coherent migration of a range of the virtual address space of the program with the `migr_range_migrate(addr, size, node)` system call. The requested memory migration is subject to the global resource management policies of IRIX. This practically means that IRIX may reject a request for migrating pages if it detects that there is not enough available memory in the target node. In general, IRIX follows a best-effort scheme for migrating pages. If the target node has insufficient free physical memory, IRIX tries to migrate the pages to a node as physically close as possible to the target node [15].

UPMLIB uses the `/proc` interface for accessing hardware reference counters. The Origin2000 memory modules are equipped with 11-bit hardware counters. There is one counter per node for each page in memory, for system configurations of up to 64 nodes. The hardware counters are memory-mapped to 32-bit software-extended counters by the operating system. When a hardware counter overflows, the system adds the contents of the counter as well as the contents of all the counters of the same page to the corresponding software-extended counters and resets the hardware counters. This implementation introduces a hysteresis of the values of the software-extended counters, compared to the actual number of references to the corresponding pages. The runtime system polls both the hardware and the software-extended counters, to cope with this asynchrony that might affect page migration decisions.

The asynchrony between hardware and software-extended counters is circumvented in the following way. Let  $n_{h,t}$ ,  $n_{s,t}$  be the contents of a hardware and the corresponding software-extended counter at time  $t$ . If  $n_{s,t} < n_{h,t}$ , the system uses  $n_{h,t}$  in the page migration criteria, since the value of the hardware counter is up-to-date with the actual number of references in this case. Suppose that  $n_{s,t} \geq n_{h,t}$ . Also, let  $n_{s,t-1}$ ,  $n_{h,t-1}$  be the values of the counters the last time the runtime system retrieved a snapshot of them. If  $n_{s,t-1} = n_{s,t}$ , the runtime system uses the formula  $n_{s,t-1} + n_{h,t} - n_{h,t-1}$ , to compute the actual number of references to the page. Note that in the same scenario, it is impossible to have  $n_{h,t} < n_{h,t-1}$ . This would mean that the hardware counter has overflowed at least once, and therefore  $n_{s,t} > n_{s,t-1}$ , which is impossible according to the original hypothesis. If the two consecutive snapshots of the counters indicate that  $n_{s,t} > n_{s,t-1}$ , the runtime system uses the formula  $n_{s,t} + n_{h,t}$  to compute the actual number of references, since  $n_{h,t}$  is the amount of references after the counter overflow in this case.

The size of the hardware page in the Origin2000 memory modules is 4 Kbytes. The page size used by the operating system to manage virtual memory is 16 Kbytes. Each virtual memory page is stored in four consecutive physical memory pages. UPMLIB combines the values of the counters of the physical memory pages that cache a virtual memory page, to compute reference rates. Furthermore, UPMLIB tries to batch mul-

multiple page migrations for consecutive pages in the virtual address space into a single invocation of the IRIX memory migration facility to reduce the runtime overhead.

The communication between UPMLIB and the IRIX kernel is realized via polling shared variables in the private data areas (`prda`) of IRIX threads, using the `schedctl()` interface. The operating system updates a flag in the `prda` of each thread, which stores the physical CPU on which the thread was scheduled during the last time quantum. UPMLIB uses this information in conjunction with hints provided by the IRIX kernel for defining the number of threads that execute OpenMP parallel/work sharing constructs. The latter can be obtained using the `mp_suggested_numthreads()` call to the IRIX parallelization runtime library. In this way, UPMLIB detects thread preemptions and migrations at the boundaries of parallel constructs to trigger the page forwarding algorithms.

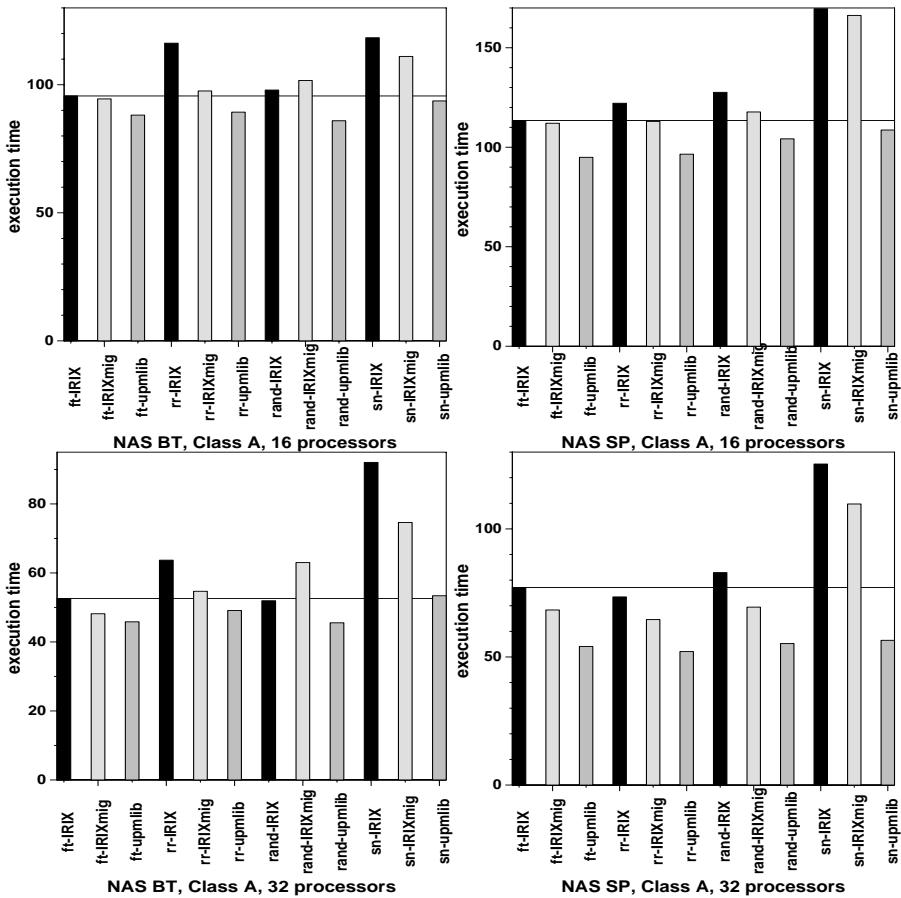
### 3.2 Mechanisms for Executing Page Migrations

UPMLIB uses two mechanisms for executing page migration algorithms. By default, the runtime system overlaps the execution of page migrations with the execution of the threads of a parallel program. We measured with microbenchmarks the average cost of a user-level page migration on the SGI Origin2000 to be equal to approximately 1–1.3 milliseconds, including the cost for reading reference counters and executing the page migration algorithm. This makes evident that UPMLIB can not execute a large number of page migrations on the critical path of the program. Therefore, the runtime system uses a separate thread, called the *memory manager*, for executing page migrations. This thread is created in sleep mode when UPMLIB is initialized and wakes up upon every invocation of UPMLIB by the OpenMP program. The memory manager executes in parallel with the application threads. This strategy works well for standalone parallel programs running on moderate to large processor scales, at which the program can gracefully sacrifice one processor for executing operating system code [6].

In loaded multiprogrammed systems in which the total number of active threads may be higher than the number of processors, the memory managers created by UPMLIB may undesirably interfere with the threads of parallel programs. To cope with this problem, UPMLIB supports also the execution of page migration algorithms from the master thread of the OpenMP program. According to the OpenMP specification, the master thread participates in the execution of parallel constructs. It is therefore important to minimize the interference between the master thread and UPMLIB code. To achieve this, the runtime system uses stripmining of the buffers that store the reference counters, in order to reduce the size of the working set size of UPMLIB and avoid erasing completely the cache footprint of the master thread. The same technique is used when the compiler uses UPMLIB for phase-driven optimization of page placement, since in this case page migrations must be performed before phase changes to ensure proper data distribution [12].

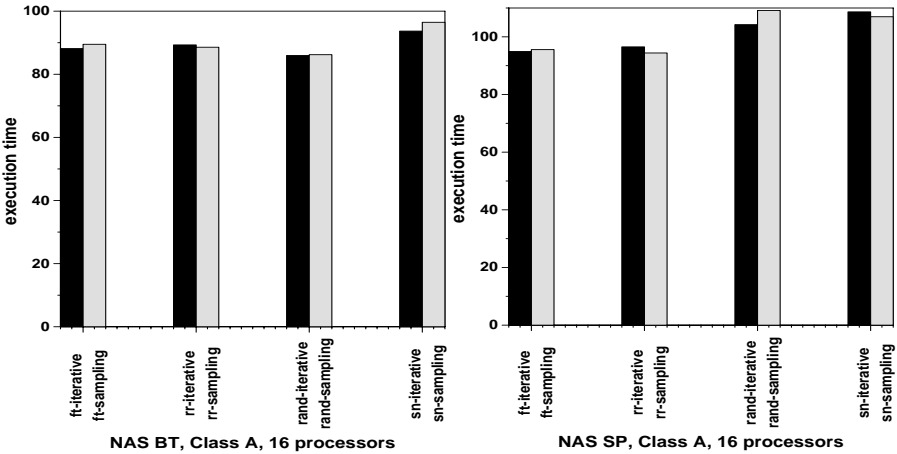
## 4 Experimental Results

In this section we provide a small set of experimental results, as case studies that demonstrate the potentials of UPMLIB.



**Fig. 3.** Performance of the NAS BT and SP benchmarks, with different page placement and migration strategies on 16 and 32 processors of the Origin2000.

Figure 3 illustrates the performance of two application benchmarks from the NAS suite, BT and SP, both parallelized with OpenMP [7]. BT is a simulated CFD application which solves Navier-Stokes equations using the Beam-Warming method. SP solves the same equations using approximate API factorization. The applications are strictly iterative, in the sense that they perform the same parallel computation for a number of time steps. Both programs are optimized by their providers, to exploit the first-touch page placement strategy, which is used by default in the Origin2000. This is done by executing a cold-start iteration of the parallel computation before the beginning of the time-stepping loop, in order to warm up the caches and place pages appropriately. The experiments were conducted on a 64-processor SGI Origin2000 with MIPS R10000 processors. Each processor had a clock frequency of 250 MHz, 32 Kbytes of primary and 4 Mbytes of secondary cache. The system had 8 Gbytes of main memory, uniformly distributed among the nodes.



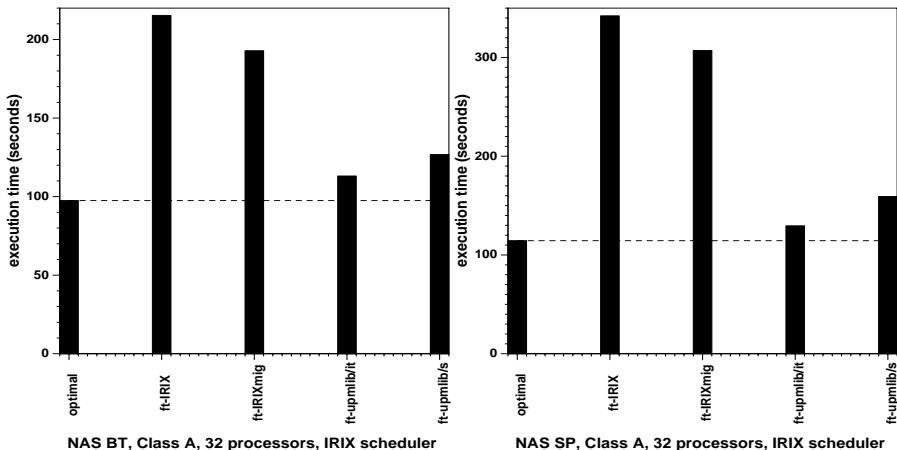
**Fig. 4.** Performance of the sampling mechanism with different page placement schemes in the NAS BT and SP benchmarks.

The charts plot the execution time of the benchmarks on 16 and 32 idle processors with four different initial page placement schemes, namely first-touch (labeled *f t*), round-robin (labeled *r r*), random (labeled *r and*) and the hypothetical worst-case placement in which all resident pages of the benchmarks are placed on a single node (labeled *sn*), thus exacerbating contention and latency due to remote accesses. The random and worst-case page placement were hand-coded in the benchmarks. For each of the three page placement schemes, we executed the benchmarks without page migration (labeled *IRIX*), with the *IRIX* page migration engine enabled (labeled *IRIXmig*), and with the *IRIX* page migration engine disabled and user-level dynamic page migration enabled by linking the codes with *UPMLIB* (labeled *upmlib*).

The primary outcome of the results is that the benchmarks exhibit sensitivity to the page placement strategy of the operating system and in the cases in which the page placement scheme is harmful, the *IRIX* page migration engine is unable to close the performance gap. For example, worst-case page placement incurs slowdowns of 1.24 to 2.10 even if dynamic page migration is enabled in the *IRIX* kernel. With round-robin page placement, the slowdown compared to first-touch ranges between 1.08 and 1.38, while with random page placement the slowdown ranges between 1.02 and 1.14.

The iterative page migration engine of *UPMLIB* brings the slowdown factor in the case of worst-case page placement down to at most 1.06. With round-robin and random page placement schemes, slowdown is less than 1.01 when the user-level page migration engine is employed. The results show that user-level page migration makes the OpenMP implementations of the benchmarks immune to the page placement strategy of the operating system and the associated problems with data locality. Furthermore, *UPMLIB* provides sizeable performance improvements (28% in the case of BT) over the best-performing page placement and migration scheme of *IRIX*.

Figure 4 illustrates the performance of the sampling mechanism of *UPMLIB*, against the performance of the iterative mechanism. The duration of the sampling in-



**Fig. 5.** Average execution time of the NAS BT and SP benchmarks, in multiprogrammed workloads executed with the native IRIX scheduler.

interval used in these experiments was 1 second and the mechanism scanned 100 pages in the hot memory areas per invocation by the runtime system. The hot memory areas were scanned by the page migration engine in a round-robin fashion. The performance of the sampling-based mechanism inferior to the performance of the iterative mechanism by at most 17%. Considering the fact that the iterative mechanism is well tuned for applications like the ones evaluated in these experiments, we can conclude that the sampling-based mechanism constitutes an effective alternative in cases in which the iterative mechanism is not applicable.

Figure 5 illustrates the results from executions of multiprogrammed workloads with the NAS BT and SP benchmarks. Each workload includes four identical copies of the same benchmark, plus a sequential background load consisting of an I/O-intensive C program. The workloads were executed on 64 processors. All instances of the parallel benchmarks requested 32 processors for execution, however the benchmarks enabled the dynamic adjustment of the number of threads that execute parallel code, via the `OMP_SET_DYNAMIC` call [13]. In these experiments, IRIX initially started all 128 threads of the parallel benchmarks, relying on time-sharing for the distribution of processor time among the programs. In the course of execution, IRIX detected that the parallel benchmarks underutilized some processors and reduced accordingly the number of threads, reverting to space-sharing for executing the workload. However, some processors were still time-shared due to the interference of the background load.

The results show the average execution time of the parallel benchmarks in the workloads with plain first-touch page placement (`ft-IRIX`), first-touch and the IRIX page migration engine enabled (`ft-IRIXmig`) and first-touch with the page forwarding heuristic, enabled with the iterative and the sampling-based mechanisms used in the page migration engine (labeled `ft-upmlib/it` and `ft-upmlib/s` respectively). The theoretical optimal execution time of the benchmarks is also illustrated in the charts. The optimal time is computed as the standalone execution time of each bench-

mark on 32 processors with the best page placement strategy (`ft-upmlib`, see Figure 3), divided by the degree of multiprocessing in the workload.

The results illustrate the performance implications of multiprocessing on the memory performance of parallel programs when their threads are arbitrarily preempted and migrated between nodes by the operating system. The average execution time of the programs is slowed down by 2.1 to 3.3 compared to the theoretically optimal execution time, when the native IRIX page management schemes are used. Instrumentation of UPMLIB has shown that the IRIX kernel performed on average about 2500 thread migrations during the execution of each workload. UPMLIB with the iterative page forwarding mechanism is very effective in dealing with this problem. The performance of the programs linked with UPMLIB is within 5% off the theoretical optimal performance. The performance of the sampling-based mechanism is inferior, although close to the performance of the iterative mechanism.

## 5 Conclusion

This paper outlined the design and implementation of UPMLIB, a runtime system for tuning the page placement of OpenMP programs on scalable shared-memory multiprocessors, in which shared-memory programming models are sensitive to the alignment of threads and data in the system. UPMLIB takes a new approach by integrating the compiler and the operating system with the page migration engine, to improve the accuracy, timeliness, and effectiveness of dynamic page migration. The experiments have shown that a smart page migration engine can obviate the need for introducing data distribution directives in OpenMP, thus preserving the simplicity of the shared-memory programming model. Moreover, dynamic page migration has demonstrated its potential as a means to provide robust performance of parallel programs in multiprogrammed environments, in which the programs can not make any safe assumptions on resource availability.

Our current efforts are oriented towards three directions: utilizing the functionality of UPMLIB in codes with fine-grain phase changes in the memory access pattern; customizing UPMLIB to the characteristics of specific kernel-level scheduling strategies; and integrating a unified utility for page and thread migration in UPMLIB, with the purpose of biasing thread scheduling decisions by page reference information to achieve better memory locality.

## References

1. E. Ayguade et.al. *NanosCompiler: A Research Platform for OpenMP Extensions*. Proc. of the First European Workshop on OpenMP, pp. 27–31. Lund, Sweden, October 1999.
2. D. Culler, J. P. Singh and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1998.
3. W. Gropp et.al. *MPI: The Complete Reference, Vol. 2*. MIT Press, 1998.
4. High Performance Fortran Forum. *High Performance Fortran Language Specification. Version 2.0*. Technical Report CRPC-TR92225. Center for Research on Parallel Computation, Rice University. January 1997.

5. G. Howard and D. Lowenthal. *An Integrated Compiler/Run-Time System for Global Data Distribution in Distributed Shared Memory Systems*. Proc. of the 2nd Workshop on Software Distributed Shared Memory, in conjunction with ACM ICS'2000. Santa Fe, New Mexico, May 2000.
6. 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, Georgia, May 1999.
7. H. Jin, M. Frumkin and J. Yan. *The OpenMP Implementation of NAS Parallel Benchmarks*. Technical Report NAS-99-011, NASA Ames Research Center. October 1999.
8. J. Laudon and D. Lenoski. *The SGI Origin2000: A ccNUMA Highly Scalable Server*. Proc. of the 24th Int. Symposium on Computer Architecture, pp. 241–251. Denver, Colorado, June 1997.
9. J. Lesvesque. *The Future of OpenMP on IBM SMP Systems*. Invited talk. First European Workshop on OpenMP. Lund, Sweden, October 1999.
10. D. Nikolopoulos et.al. *A Case for User-Level Dynamic Page Migration*. Proc. of the 14th ACM International Conference on Supercomputing, pp. 119–130. Santa Fe, New Mexico, May 2000.
11. D. Nikolopoulos et.al. *User-Level Dynamic Page Migration for Multiprogrammed Shared-Memory Multiprocessors*. To appear in the 29th International Conference on Parallel Processing. Toronto, Canada, August 2000.
12. D. Nikolopoulos et.al. *Leveraging Transparent Data Distribution in OpenMP via user-level Dynamic Page Migration*. To appear in the 3rd International Symposium on High Performance Computing, Workshop on OpenMP Experiences and Implementations. Tokyo, Japan, October 2000.
13. OpenMP Architecture Review Board. *OpenMP FORTRAN Application Programming Interface*. Version 1.1, November 1999.
14. 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.
15. Silicon Graphics Inc. *Origin2000 Performance Tuning and Optimization Guide*. IRIX 6.5 Technical Publications, <http://techpubs.sgi.com>. Accessed January 2000.
16. 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, Massachusetts, October 1996.