

# Dynamic Loop Schedulers and Memory Behavior

Ernest Artiaga, Nacho Navarro, Eduard Ayguadé and Jesús Labarta  
European Center of Parallelism in Barcelona (CEPBA)  
Department of Computer Architecture, Technical University of Catalonia (UPC)  
Jordi Girona 3-5, E-08034 Barcelona, Spain  
{ernest,nacho,eduard,jesus}@ac.upc.es

## Abstract

*Loops are the most important source of parallelism in numerical applications and OpenMP offers a work-sharing construct designed to express this source of parallelism. Nowadays, such applications may run in a wide variety of architectures, ranging from shared-memory multiprocessor systems to software DSM systems that provide a comfortable way of programming distributed systems. However, most of the loop scheduling policies available so far have been designed for shared memory environments with the aim of improving load balancing, without taking into account the impact of the cost of remote memory accesses changes in different architectures. In this paper, we analyze the behavior of some scheduling policies in a NUMA architecture and predict its behaviour with different memory access costs. From this analysis, we propose the basis for memory-aware techniques that improve the scheduling policies by reducing memory accesses while maintaining good load balancing.*

## 1. Introduction

Numerical applications devote a big portion of their time to execute loops. Such loops are a well known source of parallelism. Many scheduling techniques have been proposed to exploit this parallelism in an efficient way on shared-memory multiprocessor systems. When they were originally proposed, the main goal were to achieve good load balancing and to reduce the scheduling overhead [7]. As NUMA systems were a common thing, memory access became an important issue and schedulers took memory affinity into account: they tried to keep the computations near the location of the needed data to reduce the average memory access latency [4][10]. Nowadays, cluster-based systems are becoming widely available, and a number of DSM implementations offer a shared-memory programming model on top of such distributed systems [2][3].

Memory access costs in the available architectures differ by orders of magnitude. A cache miss and the consequent main memory access in a shared-memory system takes about nanoseconds. The same miss may cost up to a few microseconds in a NUMA architecture when having to access data in a remote memory node. On cluster architectures based on DSM mechanisms, a remote memory access

may take from several (hundreds) of microseconds up to some milliseconds, depending on the hardware support and the bandwidth of the interconnection network.

The granularity of the data being accessed also varies in the available parallel systems. The basic memory access unit in a shared-memory system is a cache line (a value around 128 bytes). Such granularity is increased up to a memory page (about 4096 bytes and above) for software DSM systems. This variation affects both the remote memory access cost (due to the amount of data being read or written) and the probability of having bad memory behavior (such as false sharing).

A desirable goal is to allow programmers to write parallel applications for all this range of platforms in the same way they did for physically shared-memory architectures, despite the effects of different memory access costs. Moreover, the same program, without modifications, should be able to run efficiently on all platforms offering a shared-memory view.

OpenMP provides a good foundation to reach such goal. OpenMP provides a shared-memory programming model through a set of directives that allow the programmer to express parallelism in its application. Then, the compiler and the runtime system are responsible for generating and using the proper code to obtain an efficient execution on the target system.

In this environment, parallelizing and scheduling loop iterations is very important to achieve good application performance. OpenMP also provides directives to let the programmer specify the appropriate scheduler policy.

The performance of the selected loop scheduling algorithm on a specific platform will be affected by the following factors:

- First, memory granularity (the size of the access unit) will affect the risk of false sharing. The probability of two processors needing data in the same page is higher than the probability of two processors needing data in the same cache line.
- Second, the remote accesses can have a weight comparable to the effects of load unbalance. So, the loop scheduler should be able to decide between balancing the load or achieving a good memory access pattern.

In this paper, we will study the influence of the memory latency on the behavior of classical loop scheduling poli-

cies. Our goal is to choose the right scheduler to make an application avoid hitting the memory wall, despite the height of such wall (i.e. the remote memory access cost).

Our study will consider a continuum range of memory access costs, from nanoseconds to milliseconds, in order to easily detect changes in the scheduler behavior.

Then, we will present the lessons learned and will make a proposal for loop scheduling policies that combine both load balancing and memory consciousness.

The paper is organized as follows: Section 2 presents some scheduling policies commonly used for numerical applications. Section 3 presents the experimental framework and methodology. Section 4 discusses the results of the experiments and section 5 presents the lessons learned and our proposals. Finally, Section 7 presents the conclusions and outlines the future work.

## 2. Loop scheduling policies

There are several policies to distribute loop iterations among different processors. Most of them may be classified as *static* (distribution of iterations is decided before the loop executes) or *dynamic* (distribution of iterations is decided during the loop execution). Some of them are oriented to reduce the scheduling overhead, others focus in achieving a good load balancing among iterations and the rest try to exploit memory affinity. In this section we will comment their main characteristics and how they can behave on a range of different memory access costs.

The goodness of a scheduler depends on its ability to get load balancing, good memory behavior, low scheduling overhead, and the weight of these factors in the application execution time. Such weight also depends on the underlying system: the effects of memory behavior will be higher in a software distributed shared memory system than in a physically shared-memory machine (due to memory access costs and granularity).

The following subsections explain the basics of both the static and dynamic loop schedulers, and also the memory-conscious schedulers, that try to combine both load-balancing with good memory behavior (usually by combining static and dynamic scheduling).

### 2.1. Static schedulers

As said before, static schedulers distribute iterations among processors before starting the loop, and the assignment is not changed during the loop execution.

The simplest static scheduling consists in dividing the iterations into the number of processors equally. This is called *Block Scheduling* or simply *Static Scheduling*. It may suffer from load unbalance if the different iterations have different costs. The Static scheduler easily exploits spatial locality. The risk of false sharing is also reduced, since only the first and last iterations assigned to a processor will access a memory unit shared with a different processor.

*Interleaved Scheduling* (or *Static Scheduling with Chunks*) is another static scheduling policy. Iterations are divided into equal sized chunks that are cyclically assigned to processors. It is aimed to reduce load unbalance: chunks

with heavy and lightweight iterations will be equally distributed among processors.

### 2.2. Dynamic schedulers

On the other side of the spectrum, dynamic policies decide which processor executes each iteration during the loop execution. Usually, the iteration space is divided into chunks; each processor takes a chunk, executes it, and asks for another chunk to execute. These techniques target load balancing, but usually with high overheads in terms of scheduling and synchronization. Memory behavior is also poor. A processor will get different iterations in different executions of the same loop, so that temporal locality is not exploited. These schedulers do not take advantage of spatial locality either, since chunks are usually small.

*Self Scheduling* (chunk size equals one) and *Uniform Sized Chunking (USC)* (chunk size greater than one) are the most representative and simple of such policies. A number of variations try to reduce scheduling overhead by grouping the iterations in different sized chunks. For example, *Guided Self Scheduling (GSS)* [7] uses this approach. It allocates large chunks of iterations at the beginning of the loop to reduce synchronization overhead, and allocates small chunks at the end to balance the load.

### 2.3. Memory-conscious schedulers

As NUMA systems became common, loop schedulers began taking memory affinity into account [4][10]. The goal was to reduce the number of remote memory accesses (or even the number of accesses to main memory). In shared memory systems, the key was getting the data into the cache during the first execution of the loop, and then distribute the same iterations to the same processors in the following executions of the loop. Then, each processor would access its local cache, instead of the (possibly remote) main memory.

A common approach for these schedulers is to use a static scheduler during the first part of the loop execution, combined with a dynamic scheduler in the last part of loop execution to compensate possible load unbalance.

It is important to note that in high remote access cost systems (like software DSM), the convenience of using dynamic schedulers to compensate load unbalance must be studied carefully, since the cost of bad memory behavior will be higher than in shared-memory systems.

One of the basic memory-conscious schedulers is *Affinity Scheduling* [4]. The goal is having a processor take the same iterations (and access the same memory) each time the loop is executed. The affinity scheduler also achieves load balancing: when a processor finishes all its assigned work, it may help another processor by stealing some of its chunks in a dynamic way. There are algorithms specifically designed to work on SDSM (software DSM) platforms (like ABS[9] and Adaptive Affinity Scheduler [11]) based on Affinity scheduler.

Another variation of Affinity scheduling tries to reduce the dynamic portion every time the loop is executed. *Dynamically Partitioned Affinity Scheduling (DPAS)* [10] starts as Affinity Scheduling, but at the end of the loop, each processor remembers how many iterations has actu-

ally executed (the initial ones plus/minus the stolen ones). As the next loop execution begins, the initial distribution of iterations is adjusted according to the actually executed iterations. Notice that this redistribution does not imply that the same iterations will be executed.

Finally, *Feedback-Guided Dynamic scheduling* [1] also tries to achieve good load balancing from an initial static scheduling distribution. Instead of having a dynamic portion and adjusting iteration distribution based on the number of chunks executed, FGDS measures the execution time used by each processor and uses that to estimate the cost of iterations. Then, the size of chunks for the next execution of the loop are recomputed based on this estimation.

### 3. Experimental framework

#### 3.1. Hardware

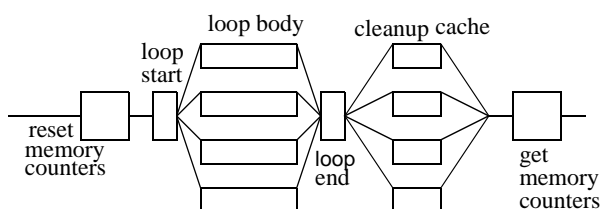
The benchmarks have been run on a SGI Origin 2000 with 64 processors, running Irix 6.5.6. The system is based in the MIPS R10000 processor, and it consists of 32 interconnected nodes. Each node has 2 processors with their respective secondary cache memories (4 MB per processor). The system has a distributed architecture, but it provides global addressability from any processor in any node to all memory.

The memory is divided into physical pages of 4KB. Each physical page has a set of 32 hardware counters to count the number of memory references from each node to that page. Such counters are accessible at user level via the `ioctl` system call. In this paper, we will use them to study and predict the memory behavior of the different schedulers.

#### 3.2. Methodology

All benchmarks have been executed on the SGI Origin 2000 inside a `cpuset`. This procedure ensures that the application threads will run on a specific set of processors and no other application can use them while the benchmark is running. We also wired each thread to a single processor to avoid threads jumping from one processor to another due to operating system scheduling decisions. This also allows us to map the events registered by hardware counters (cache misses, page references, etc.) to an application thread.

Loops have been parallelized using OpenMP directives. Then, we use dynamic interposition tools [8] to instrument the execution and dynamically change the loop scheduler and take some of the measurements. The resulting code has the following structure:



The hardware counters for the accessed memory areas are reset before starting the loop; then, multiple threads execute the loop body. At the end, individual execution time of each thread is recorded, as well as the value of the hardware counters. It is important to note that these memory access counters do not imply any additional overhead, since they are implemented in hardware. Eventually, each processor flushes its cache to update the hardware counters and then, the memory access counters are read.

In this study, we only consider read-write data. We consider that each processor can maintain a copy of read-only data in cache or local memory, so that it will produce, at most, a single remote access during the first access. Hence, its impact on performance will be much smaller than the effect of read-write data.

#### 3.3. Benchmarks

We have used, as benchmarks, some syntetic applications that are representative of loop codes found in real applications. Each benchmark focusses in a certain aspect (affinity, load unbalance, etc.). The codes are based on the benchmarks used in [4] and [10].

We have used the following benchmarks:

- ADJCONV (Adjoint Convolution). The loop iterations are highly unbalanced (load distribution has a triangular shape). Memory write operations show high affinity, though read memory operations do not show affinity at all. The problem size is 256x256 and the loop is repeated 40 times.
- JACOBI implements the jacobi algorithm to solve a system of linear equations. The input is based in a sparse coefficient matrix, so that only non-zero elements are explicitly stored. Such matrix is accessed read-only and there are write accesses to 2 vectors: the solution  $x$  vector and another one to store partial sums. Access to such vectors show high memory affinity. The work to generate them is very unbalanced, though the load pattern is repeated in successive executions of the parallel loop. The problem size is 16384x16384 and the main loop is executed 100 times.
- SOR (Successive Over-Relaxation). The parallel loop in this algorithm is well balanced (all the iterations take about the same time). Write accesses to the result matrix show certain affinity. Read-only accesses to the source matrix are not considered, as commented in section 3.2. Problem size is 1024x1024 and the main loop is executed 100 times.
- TRANCLOS (Transitive Closure). This code shows high load unbalance, as each iteration may take  $O(1)$  or  $O(N)$  time ( $N \times N$  being the size of the input matrix); moreover, the load pattern changes for different executions of the loop. It also shows some memory affinity, since each iteration tends to access the same areas of the matrix. We have used a matrix of 512x512 elements and the main loop is repeated 512 times.

- GAUSELIM (Gaussian Elimination). This application shows low load unbalance and some memory affinity. The lower loop limit varies in the different executions of the loop. The matrix size is 1024x1024 and the loop is executed 512 times.

The parallelization of the loops uses OpenMP directives and the 'runtime' scheduling type for the loop. This scheduler type allows us to specify which scheduler to use during the execution time, without having to modify the benchmark or recompile it.

OpenMP provides the following scheduler classes for loops: Static (Block Scheduling), Static-with-chunk (Interleaved), Dynamic (Self-Scheduling and USC) and Guided (for Guided Self Scheduler). We have implemented a library which adds new loop schedulers to those defined by OpenMP, and they can be activated through the runtime schedule directive. The following subsection describes such schedulers.

### 3.4. Schedulers

We have implemented some loop schedules in a runtime library in order to execute the benchmarks with them and compare the results. The library also contains some instrumentation code to obtain per-thread execution times and the memory access behavior.

Our library provides the following loop schedulers:

- **STATIC**: iterations are divided into equally sized  $p$  chunks ( $p$  being the number of threads).
- **INTER $xx$** : this is a implementation of the Interleaved (Static-with-Chunks) scheduler. The number  $xx$  represents the size of chunk.
- **AFFINITY**: implements an statically partitioned affinity scheduler, as described in [4].
- **DPAS**: implements a dynamically partitioned affinity scheduler, as described in [10].
- **ADJUST**: implements a variation on the Feedback Guided Dynamic Scheduler described in [1]. We use this scheduler with two different variations. The first one is the basic ADJUST, which tries to do a single-shot balancing after the first execution of the

loop. The second variation, ADJUST+L, keeps taking time measurements and using them as feedback to do further adjustments in the iteration distribution.

- **SELF $xx$** : implements a Uniform Sized Chunk scheduler. The number  $xx$  represents the size of chunk.

## 4. Experimental results

Results presented in this paper refer to executions using 8 processors. When possible, parameters of the scheduling algorithms have been tuned to obtain maximum performance. For example, the chunk size for the self-scheduler is 8 iterations for AdjConv and SOR, and 32 iterations for GausElim, Jacobi and TransClos. The Static-with-Chunk scheduler only made real sense for AdjConv and Jacobi. In both cases the we used a chunk size of 512 iterations.

Figure 1 shows the normalized execution time of each benchmark and scheduler on the SGI Origin 2000. Load unbalance is one of the main causes of performance degradation when scheduling the iterations in a parallel loop. This unbalanced behavior happens in the loops of AdjConv, Jacobi and TransClos and clearly affects the Static scheduler in AdjConv and Jacobi: its results almost double the best ones. Nevertheless, the unbalance does not degrade the performance in TransClos. As expected, the Static Scheduler obtains good performance in balanced applications (GausElim and SOR).

The reason for this behavior is in how the load is distributed among the iterations. For Adjconv, the load has a triangular shape. First iterations are more expensive than the last ones; so, first threads take longer to execute its chunk. At the other end, Transclos has long and short iterations distributed uniformly, so that each chunk gets the same amount of both types. This makes the Static scheduler behave well.

Using an Interleaved scheduling with an adequate chunk size may improve the distribution of work. This is true for Adjconv and, in general, for loops with a regular triangular shape. The chunk size of 512 iterations was chosen to fit chunk accesses to a single memory page and avoid possible memory conflicts. Results show that this size is enough to equally distribute light and heavy iterations among the dif-

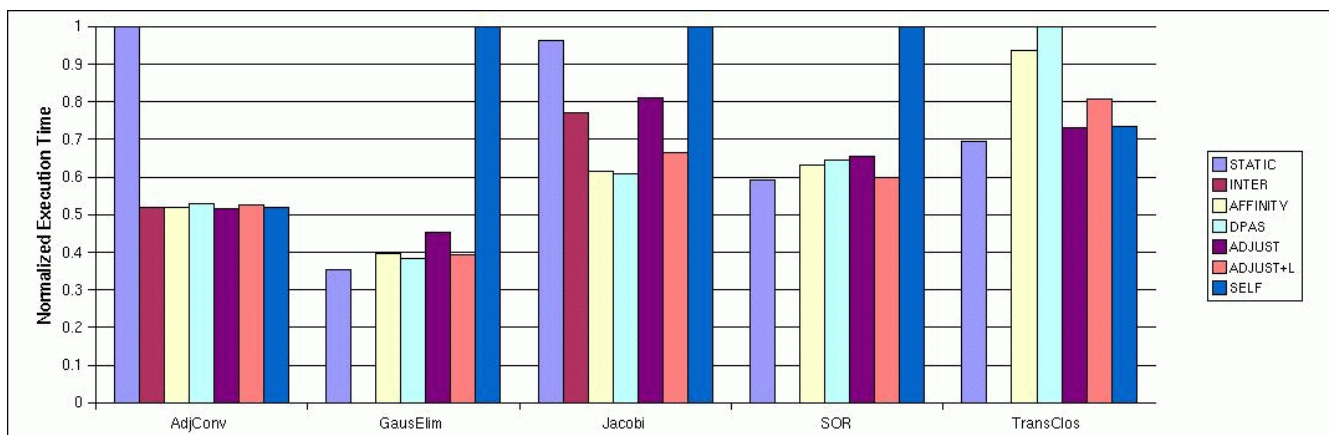


Figure 1. Normalized execution times in the SGI Origin 2000 (8 cpus)

ferent processors. This is not the case for Jacobi. The value of the chunk size avoids false sharing but it is too high to improve load balancing.

Affinity scheduler and DPAS obtain good results in all the benchmarks except TransClos. In general, combining an initial static phase with a final dynamic phase to compensate load unbalance is a good policy for a low remote memory access cost system. Nevertheless, the algorithm can make a thread steal too much work from another thread if the load pattern is too irregular. This is the cause of poor results for TransClos.

DPAS does not improve the results in GausElim because its loop changes its limits each time the loop is executed; so, our implementation of DPAS has no chance to 'learn' the behavior of the loop and reuse the data. Adjust scheduler suffers from the same problem, and it also fails to improve the Static results. DPAS also fails to learn the behavior of TransClos because the load of each iteration changes each time the loop is executed, so that history is not useful.

Adjust scheduler also gets good results with both balanced and unbalanced loops. Using continuous adaption (Adjust+L) slightly improves the results. Using big chunks also allows it to take advantage of unbalanced loops with uniformly distributed heavy and light iterations (such as TransClos).

It is interesting to note that Adjust, and also Affinity and DPAS suffer no penalty when used for balanced loops. In such circumstances, the schedulers derive to an almost static behavior. Moreover, the programmers do not have to worry about fine tuning scheduler parameters such as the chunk size, letting them concentrate on the application problem solving.

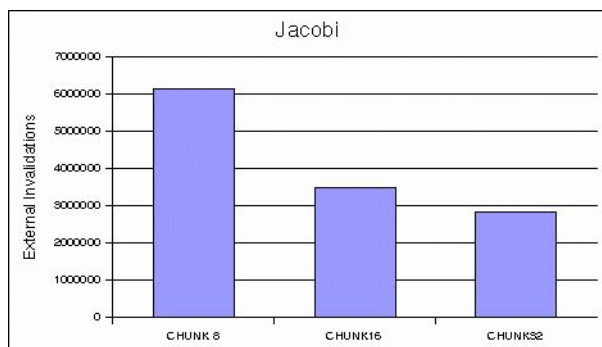


Figure 2. Number of cache line invalidations for Self-Scheduler vs. size of chunk

Finally, Self-Scheduler achieves good results in AdjConv and TransClos, but very poor ones in the rest of benchmarks. The basic reason is that Self-Scheduler is prone to produce bad memory behavior (no reuse of data and false sharing). In GausElim, TransClos and Jacobi we observed slight improvements as we increased the chunk size. This usually reduces the number of invalidations caused by false-sharing at the cache-line level, as shown in Figure 2 for Jacobi.

In some cases, the inadequate memory alignment of data structures may introduce a noticeable performance degradation. This causes chunks of iterations to use a memory area that begins and ends in the middle of a memory

unit (a cache line or a page). So, it will produce access conflicts with the contiguous chunks if they are processed by different threads. Improper alignment is commonly produced by allocating data structures in the stack or by using *malloc()*-like calls. Table 1 shows the difference in remote accesses due to memory alignment in Jacobi.

Table 1. Alignment of read-write data structures in Jacobi

Scheduler	Remote Accesses (Not Aligned)	Remote Accesses (Aligned)
Static	2503	0
Interleaved (512)	16514	2364
Affinity	1646089	1731400
DPAS	789281	704516
Adjust	6948	2180
Adjust+L	6694	4064
Self-Scheduler (32)	660947	122024

#### 4.1. Increasing the memory wall

Despite the usually good memory behavior, load unbalance is a handicap for Static scheduler, as seen for AdjConv and Jacobi. The poor balancing leaves room enough for other schedulers to get better results, even though they may suffer from a bad memory behavior.

In this subsection, we use a simple model to study to analyze the influence of the remote memory latency on the scheduler. In this model, we first compute the number of remote accesses from a node as the number of write accesses from that node multiplied by the probability of the previous write access being done by a different node. We obtain that value in the following manner: let  $R_p$  be the total number of write references from all nodes to the page  $p$ ,  $R_{pn}$  the number of write references from node  $n$  to page  $p$ , then we compute the number of remote memory accesses due to remote modifications  $I_n$  of node  $n$  as:

$$I_n = R_{pn} \cdot \frac{R_p - R_{pn}}{R_p}$$

Both  $R_p$  and  $R_{pn}$  are obtained from the SGI hardware counters. As we do not have the exact sequence of memory references to a page, we assume that they are uniformly distributed in time (although this is not the real behavior). Then, we add to each thread's actual execution time an overhead proportional to the number of its remote accesses. This estimation is approximate because it does not feed-back to the loop scheduler the time that the remote accesses would have taken, which may have had an influence in the dynamic part (for example, number of iterations stolen) of some of the loop schedulers.

Figure 3 shows the results of the applying this model in two of the benchmarks (AdjConv and Jacobi).

The Interleaved scheduler reduces the impact of load unbalance in AdjConv. Nevertheless, it suffers from a slightly worse memory behavior: as the size of chunks decrease, there are more memory units shared by chunks executed by different processors. As a consequence, we can observe that Interleaved execution time smoothly increases with the remote memory access cost.

We can observe that Interleaved cannot always compensate for the load unbalance of Static scheduler. This is the case of Jacobi, where Interleaved improves the execution time but cannot match the initial results of more sophisticated load balancing algorithms, such as Affinity, DPAS or Adjust. Nevertheless, it is interesting to note that Interleaved scheduler becomes competitive (compared to Affinity and DPAS) when remote access time is above 300 microseconds, due to its better memory behavior.

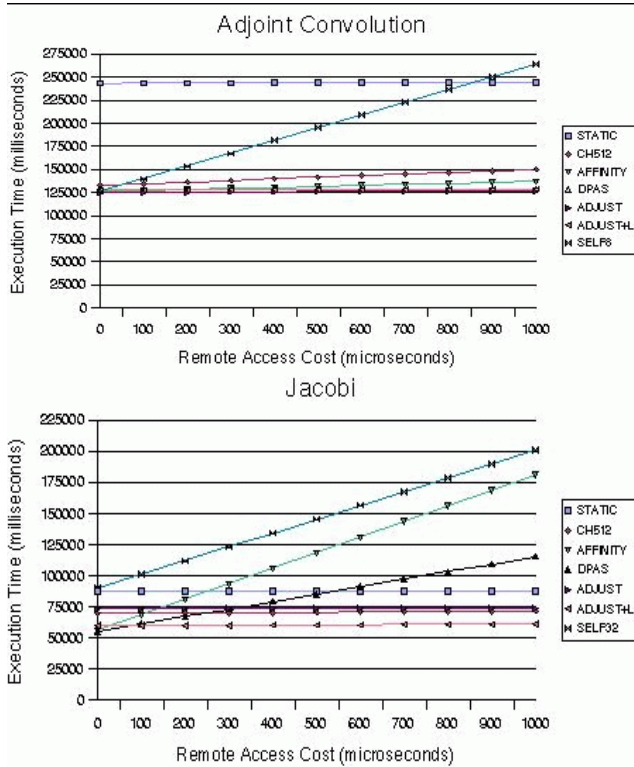


Figure 3. Execution time vs. remote access cost

Affinity scheduler shows good results for relatively low cost memory accesses (the environment for which it was designed). But its dynamic phase loads the balance at the cost of producing a high number of remote accesses. This produces a very steep curve as we increase the time to access remote memory.

In general, it is clear that dynamic policies (or phases) are less tolerant with high remote access costs. This kind of mechanisms produce random distribution of chunks and, consequently, random accesses to memory, so that the chances of having different processors trying to access the same memory unit is very high. Moreover, iteration distribution is different every time the loop is executed, so that data in the cache or local memory can be rarely reused in the next execution of the same loop or other loops accessing the same data structures.

DPAS introduces the ability to 'learn' from previous history of the loop execution. This mechanism makes DPAS get closer to static behavior each time the loop is executed (so that the dynamic phase is successively reduced). This drastically improves the results of Affinity scheduler for

medium to high remote memory access costs, as can be observed in Figure 3.

Adjust scheduler combines both good memory behavior (so that it usually shows an almost-flat curve when the remote memory access time is increased) with usually good load balancing. It is interesting to note that this scheduler converges quite fast (the results of Adjust, which tries to balance the load just after the first loop execution, are very similar to the results of Adjust+L, which tries to adapt and rebalance after each execution of the loop).

Such properties make an Adjust-like scheduler convenient for a wide range of systems. Compared with the Interleaved scheduler, we can observe that it has a slightly better memory behavior in AdjConv, and better load balancing in Jacobi.

Finally, our experiments show that Self-Scheduler is not adequate for systems with non-negligible memory access costs. Random accesses generated by Self-Scheduler produce a lot of memory conflicts, which are translated into a steep execution time curve as the memory access cost increases.

Increasing the memory access costs up to 10 milliseconds shows the same trends above. For example, Figure 4 shows the schedulers behavior for Jacobi. We can see that for high remote access costs (corresponding to SDSM without high bandwidth interconnection networks), Affinity and DPAS are not adequate.

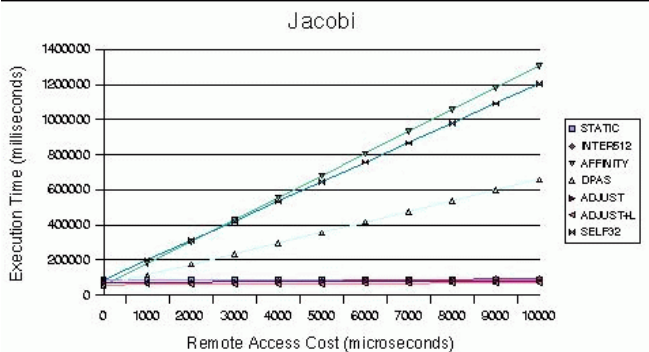


Figure 4. Behavior for very high remote access costs in Jacobi.

## 5. Proposal

From the results of our experiments and model, we can conclude that an efficient loop scheduling policy should take into account the effects of the memory behavior as a base for scheduling decisions. This is specially important when dealing with high remote memory access costs (as those in a software DSM system). Notice that the conclusions drawn from the model are quite different from the ones derived from the execution on the Origin 2000.

In such systems, fitting the chunk limits to page boundaries would reduce false sharing within a memory page and, therefore, it would reduce the number of remote memory accesses due to memory conflicts.

Following this idea, we propose modifications to scheduling algorithms to improve their behavior on high access cost memory systems such as DSM. Our proposal consists

of making the scheduler take its decision based on the memory areas accessed. We have applied this idea to Adjust scheduler.

Adjust and Adjust+L achieve good load balancing with low overhead. Its lack of dynamic portions also favours it for systems with high remote access costs. Nevertheless, dividing the iteration space at arbitrary points can make different processors access the same memory page - and cause false sharing.

Our modification to the Adjust algorithm consists of partitioning the loop iterations based on timing, and then moving the partition limits to the nearest iteration with a page boundary. Our prototype implementation for the SGI Origin 2000 used information obtained from the hardware counters to estimate the iterations corresponding to memory page boundaries.

The reduction in remote accesses when running Jacobi using 8 processors is shown in Table 2. As we can observe, properly aligning the data structures to a page boundary improves the results.

Table 2. Number of remote accesses

	Adjust+L	Adjust+Align
Jacobi 16384x16384	4064	1280
Jacobi 16384x16384 (not aligned)	6694	3340

Of course, such movement may suppose some load unbalance. But this should not be noticeable in most cases, because the adjustment is very small (half the iterations that fit in single memory page, at most).

Information for automatic detection of page boundaries must be provided by the underlying system, either by hardware counters (similar to the ones in the SGI Origin 2000) or by the runtime system in case of software DSM (the system should notify when a page fault is produced in a certain memory area, and the scheduling code would take note of the current iteration).

The relation of page boundaries and iterations may also be provided by the compiler, or even the application by means of directives.

In case the loop accesses different memory areas with different alignments, information should be provided to determine which one is the most important for performance (for example: an array that is written would have a greater impact on number of memory conflicts and performance than another one that is just read).

The auto-align mechanism can be used to improve not only the Adjust-like schedulers, but also other schedulers that use variable chunk sizes (such as Guided Self-Scheduler, Affinity or DPAS).

We can also modify the Interleaved scheduler to be memory-conscious. The basic idea was proposed in [5] with the goal of reusing data in the cache when using Interleaved scheduler and the loop limits change at each iteration of the loop (as in GausElim).

Authors have recently proposed the possibility of defining schedules in OpenMP and reuse them later in the execution of a new instantiation of the same loop or even a different loop [6]. A schedule can be defined the first time the loop is executed or can be derived from the mechanism

proposed in this section (combining load balancing and good memory behavior).

## 6. Conclusions and future work

In this paper we have studied the influence of the remote memory access cost in the behavior of different loop schedulers. We are specially interested in the portability of such scheduling mechanisms to a whole range of memory latencies, from shared-memory systems to software DSM systems.

The study has been oriented to platforms providing a shared-memory programming paradigm, without special semantics (such as relaxed consistency) or platform-specific modifications of the application code.

We have shown the importance of both good load balancing and good memory behavior to achieve good performance in a large range of systems, from shared-memory machines to software DSM systems.

Both goals can be obtained by using static schedulers such as those based on FGDS (Adjust). Dynamic-based schedulers usually achieve good results for systems with reduced remote memory access costs, but they suffer from their random memory access pattern as the memory access costs are higher.

Adjust scheduler achieves competitive load balancing while keeping a good memory behavior for a large range of memory access costs. Moreover, their results are comparable to the best scheduler for all the benchmarks examined. Such features make it a good choice as a portable scheduling mechanism.

Reusing information from previous executions of the loops also improves the results. Adjust scheduler achieves good load balancing by using this method, and DPAS slightly improves its memory behavior.

It is also interesting to note that adaptive schedulers such Adjust and DPAS have few overhead when applied to balanced loops. Moreover, the programmers do not have the responsibility of adjusting parameters like the chunk size, which can vary depending on the underlying system.

Currently, we are evaluating the different scheduler algorithms in a real DSM system. Some of the predictions in this paper have already been validated on a DSM implementation.

## 7. Acknowledgments

This work has been supported by the Spanish Ministry of Science and Technology and the European Union FEDER program under contract TIC2001-0995-C02-01, and by the European Center for Parallelism of Barcelona (CEPBA).

## 8. Bibliography

- [1] J. Mark Bull, "Feedback Guided Dynamic Loop Scheduling: Algorithms and Experiments," Proceedings of EuroPar'98, 1998.
- [2] W. Hu, W. Shi, and Z. Tang, "Jiajia: An Svm System Based on a New Cache Coherence Protocol," Proceedings of the High Performance Computing and Networking Europe HPCN'99, Apr. 1999.

- [3] P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," Proc. of the Winter 1994 USENIX Conference, pp.115-131, Jan. 1994.
- [4] E.P. Markatos and T.J. LeBlanc, "Using Processor Affinity in Loop Scheduling on Shared Memory Multiprocessors," IEEE Trans. on Parallel and Distributed Systems, vol. 5, no. 4, pp. 379-400, 1994.
- [5] D.S. Nikolopoulos, E. Ayguadé, J. Labarta and T.S. Papatheodorou, "The Trade-Off between Implicit and Explicit Data Distribution in Shared-Memory Programming Paradigms," Proc. of the 15th ACM International Conference on Supercomputing (ICS'2001), June 2001.
- [6] D.S. Nikolopoulos, E. Artiaga, E. Ayguadé and J. Labarta. Exploiting Memory Affinity in OpenMP through Schedule Reuse. ACM SIGARCH Computer Architecture News, vol. 29, no. 5, pp. 49-55. December 2001.
- [7] C.D. Polychronopoulos and D.J. Kuck, "Guided self-scheduling: A practical scheduling scheme for parallel supercomputers," IEEE Trans. Comput., vol. C-36, no. 12, pp 1425-1439, Dec. 1987.
- [8] A. Serra, N. Navarro and T. Cortés, "DITools: Application-level Support for Dynamic Extension and Flexible Composition", Proc. of USENIX Annual Conference, June 2000.
- [9] W. Shi, Z. Tang and W. Hu, "A More Practical Loop Scheduling for Home-based Software DSMs," Proceedings of ACM-SIGARCH Workshop on Scheduling Algorithms for Parallel and Distributed Computing, June 1999, Greece.
- [10] S. Subramaniam and D.L. Eager, "Affinity Scheduling of Unbalanced Workloads," SuperComputing'94 Conference Proceedings, 1994
- [11] Y. Yan, C. Jin, and X. Zhang, "Adaptively Scheduling Parallel Loops in Distributed Shared Memory Systems, " IEEE Trans. on Parallel and Distributed Systems, vol. 8, no. 1, pp. 70-81, Jan. 1997