Data Distribution and Loop Parallelization for Shared-Memory Multiprocessors *

Eduard Ayguadé, Jordi Garcia, M. Luz Grande and Jesús Labarta

Computer Architecture Department, Polytechnic University of Catalunya
cr. Gran Capità s/núm, Mòdul D6, 08034 - Barcelona, Spain

Abstract. Shared-memory multiprocessor systems can achieve high performance levels when appropriate work parallelization and data distribution are performed. These two actions are not independent and decisions have to be taken in a unified way trying to minimize execution time and data movement costs. The first goal is achieved by parallelizing loops (the main components suitable for parallel execution in scientific codes) and assign work to processors having in mind a good load balancing. The second goal is achieved when data is stored in the cache memories of processors minimizing both true and false sharing of cache lines. This paper describes the main features of our automatic parallelization and data distribution research tool and shows the performance of the parallelization strategies generated. The tool (named PDDT) accepts programs written in Fortran77 and generates directives of shared memory programming models (like Power Fortran from SGI or Exemplar from Convex).
Keywords: High Performance Compilers, Loop Parallelization, Static and Dynamic Data Mappings, Cache Behavior, Shared Memory Multiprocessors

1 Introduction

Parallelization and data distribution are two topics closely related when parallelizing loops for cache-coherent shared-memory parallel systems. In these systems, cache miss penalties can be significantly large and false sharing, invalidations and excessive data replication can have negative effects in performance. In some cases, these effects can easily offset any gain due to parallel execution.

Most current shared-memory compilers choose a loop in each nest for parallelization, and it is interchanged as far out as data dependence analysis allows. Inner loops are strip-mined and blocked to exploit all possible data reuse in the processor cache. Iterations in each parallel loop are distributed across the parallel threads according to a fixed scheme. Some compilers also ensure that each major data structure in the program is aligned on a cache line boundary and make the contiguous dimension of an array (i.e., the first dimension in Fortran) an integer multiple of a cache line. This is useful to avoid false sharing of cache lines so that each processor works with complete cache lines.

* This research has been supported by the Ministry of Education of Spain under contract TIC-429/95 and by the CEPBA (European Center for Parallelism of Barcelona).

Ayguadé, E. [et al.]. Data distribution and loop parallelization for shared-memory multiprocessors. A: International Workshop on Languages and Compilers for Parallel Computing. "Languages and Compilers for Parallel Computing, 9th International Workshop, LCPC'96: San Jose, California, USA, August 8–10, 1996: proceedings". Berlín: Springer, 1996, p. 41-55. ISBN 978-3-540-69128-0.

The final authenticated version is available online at https://doi.org/10.1007/BFb0017244

Some researchers [AL93] have focussed on better determining which loop to parallelize with the purpose of obtaining maximum parallelism while minimizing sharing of cache lines (true sharing). They analyze data and computation decomposition without regard to the original layout of the data structures. A more recent work [AAL95] proposes to enhance spatial locality, reduce false sharing (access to different data items co-located on the same cache line) and conflict misses among accesses to the set of data assigned to each processor. This is done by applying some data transformations making data accessed by each processor contiguous in the shared address space. [JE95] have also proposed algorithms to transform data layouts to improve memory performance; they analyze per-process shared data accesses in parallel programs, identify data structures that are susceptible to false sharing and choose an appropriate layout transformation to reduce the number of false sharing misses. These data layout transformations require that all accesses to the arrays in the entire program use the new layout; programming languages (such as Fortran) can make these transformations difficult and the compiler has to guarantee that all possible accesses are updated accordingly and optimized.

In the past years, other researchers have targeted their efforts to automatic data distribution for distributed-memory multiprocessors [BCG+95, AGG+95, KK95, SSGC95], according to the array access patterns and parallel execution of operations within computationally intensive phases. The objective is to specify the mapping for the arrays used in these computational phases, and it can be either static or dynamic. In a static mapping, the layout of the arrays does not change during the execution of the program; in a dynamic mapping, remapping operations are performed in order to change the layout of arrays in different computational phases.

A basic observation of this paper is that this technology developed for distributed memory compilers is useful for shared memory architectures in which each processor has access to a high-capacity private cache (for instance, 4 Mbyte in each processor of a R8000 SGI Power Challenge or between 512 Kbyte and 16 Mbyte in each processor of a R10000 SGI Power Challenge architecture [SGI96]). In these systems, the cache behaves as an attraction local memory that stores data referenced by the processor. Trying to minimize true and false sharing reduces data motion through the interconnection network. The techniques we have developed represent the application of the owner computes rule, frequently used in distributed-memory systems, to shared-memory machines.

In a parallel loop, a chuck of iterations is assigned to each processor. The execution of this chunk will bring any remote data to its cache. Notice that data remapping is implicitly done by the caching mechanism itself. We propose to parallelize loops taking into account the data that is stored in the private cache of each processor, either because it has been previously computed or fetched in other loops, or that needs to be stored in the cache because it will be useful in the following loops. PDDT keeps track of the array sections that are accessed during the execution of the different computational phases in an application in order to decide, with a global view, the parallelization strategy for each loop. This is done

by analyzing the reference patterns inside computational phases and predicting the cache behavior that different parallelizations would imply. The generation of code for the target shared-memory programming models makes intensive use of well known techniques, such as loop tiling and loop limit adaptation to partition the iteration space, loop interchange to reduce the overhead of parallel thread creation and improve spatial locality, and parallel synchronized execution of dependent loops to minimize execution time.

In cache-coherent shared-memory systems, false sharing might also introduce additional data motion. Since data is transferred in cache lines (for instance, 128 bytes long in SGI Power Challenge multiprocessors), different processors may share the same cache line and never access to the same data items. Every time a processor writes a data item in the line, other copies of the same line are invalidated. When another processor re-uses a data item (col-located on the same cache line), the item may no longer be in its cache due to the access by the other processor. Therefore, spatial locality may be lost and additional data movement may happen. PDDT also addresses the problem of minimizing false sharing by synchronizing the access to cache lines shared by different processors in parallel loops. In addition to that, PDDT also pads the contiguous dimension of arrays to make it multiple of cache line size and aligns major data structures to cache line boundaries.

Other techniques oriented to the optimization of code for uniprocessor cache performance are left to the native compiler of the target parallel machine and they are out of the scope of this paper.

PDDT is a research tool in the sense that it is flexible to specify machine dependent characteristics and to specify different compilation options and strategies. In addition to automatic parallelization, PDDT is also a performance prediction tool that may help the user in the task of writing parallel code for the target machine; it accepts directives in the source program which narrows the search space of solutions and provides the user with information about the behavior of the program.

The paper is organized as follows. Section 2 shows the main aspects that are considered in PDDT to generate parallelization strategies based on data distributions. Section 3 summarizes the main modules in PDDT that perform the parallelization process. More details about specific modules can be found elsewhere [AGG+94, AGG+95]. Section 4 evaluates the parallelization strategies explored by PDDT and compares them against the ones generated by a commercial compiler. Finally, Section 5 states our conclusions and summarizes future work.

2 Parallelization for Coherent Caches

In this section we show the feasibility of using information about data distribution to drive the parallelization process of a program in a coherent cache-based multiprocessor system. Keeping track of data motion among the caches in the system is useful to decide parallelization strategies in which both computation

time and data movement costs are minimized. We also show further optimizations to reduce the negative effect caused by dependences and false sharing. In this section, we use an excerpt of the Alternate Direction Implicit (ADI) integration kernel[2]. The full kernel is evaluated in Section 4.

2.1 Using Data Mappings

The behavior of coherent caches is modeled when flowing from one computational phase to another within the execution of a procedure and inter-procedurally. We can model either distributed network caches (like in the Globally Shared Memory Convex SPP systems [Con94]) or private caches in bus based symmetric multiprocessor systems (like the Power Challenge SGI systems [SGI96]). To perform this modeling, we assume that capacity and conflict misses never happen[3].

In this section we analyze the two phases ($P4$ and $P7$) shown in Figure 1.a, and assume for each of them a parallelization strategy: the i loop in phase $P4$ and the j loop in phase $P7$ are fully parallelized. Figure 1.b shows the elements of the arrays that will be stored in the private cache of one of the processors (assuming 4 processors). Notice that, due to the parallelization of the inner i loop in phase $P4$, a set of consecutive rows of arrays a, b and x will be stored in the cache of each processor. Since arrays b and x are written, these elements will be owned by it and other copies will be invalidated, if existed. In phase $P7$, the outer j loop is parallelized. After executing this phase, each processor will own a set of columns of arrays b and x (since the rows that it owned have been invalidated by the other processors) and will have in its cache a set of rows and columns of array a. When these phases are executed again (because of the outer iterative *iter* loop), processors either have a set of rows or columns of arrays b and x, and a set of rows and columns of array a.

The parallelization strategies that we consider lead to cache contents that can be characterized in terms of HPF-like array alignments and distributions. So for instance, for phase $P4$ and the first iteration of the *iter* loop, one could say that arrays a, b and x are perfectly aligned and distributed across the caches in a $(BLOCK, *)$ way. In phase $P7$ and for arrays b and x, a $(*, BLOCK)$ distribution characterizes the contents of the cache. However, for array a the contents of the cache can not be characterized with a single distribution function; instead, one could say that it is the union of two distribution functions: $(BLOCK, *) \cup (*, BLOCK)$.

To estimate data movement costs, PDDT detects that the three arrays are moved when transitioning from phase $P4$ to phase $P7$ in the first iteration. After that, and for the rest of iterations, only arrays b and x are moved when alternating between these two phases; array a is not moved because each processor holds the rows and columns it needs to perform the computations.

[2] We use two different data sets: small (NUM=64 and MAXITER=1000) and large (NUM=256 and MAXITER=100).

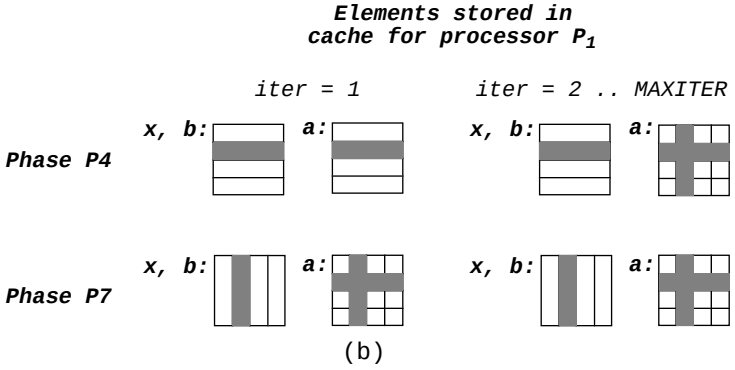
[3] This assumption holds along the paper; some comments about handling these misses and including them in the parallelization process are given in Section 5.

```
   program adi
     double precision x(NUM,NUM)
     double precision a(NUM,NUM), b(NUM,NUM)

     do 10 iter = 1, MAXITER
        do 4 j = 2, NUM                                    Phase P4
          do 4 i = 1, NUM
            x(i, j) = x(i, j) - x(i, j - 1) * a(i, j) / b(i, j - 1)
            b(i, j) = b(i, j) - a(i, j) * a(i, j) / b(i, j - 1)
4       continue

        do 7 j = 1, NUM                                    Phase P7
          do 7 i = 2, NUM
            x(i, j) = x(i, j) - x(i - 1, j) * a(i, j) / b(i - 1, j)
            b(i, j) = b(i, j) - a(i, j) * a(i, j) / b(i - 1, j)
7       continue
10   continue
   end
```

(a)



(b)

Fig. 1. (a) Excerpt of the ADI kernel. (b) Cache contents after executing phases $P4$ and $P7$.

For these two phases two other parallelization strategies would be possible: to execute phase $P4$ sequentially (because of data dependences) and phase $P7$ in parallel, or vice versa. If a single processor executes the sequential phase, then this processor will perform additional data movements and invalidations with the associated overhead. In the next section it is shown how PDDT minimizes the effect of this movement. Table 1 compares the execution time for the three parallelization strategies and for different number of processors (P). Notice that in this case, it is better to execute both phases in parallel.

	NUM=64			**NUM=256**		
	P=2	**P=4**	**P=8**	**P=2**	**P=4**	**P=8**
$P4_{par}$ and $P7_{par}$	3.89	2.10	2.71	6.18	3.32	1.74
$P4_{par}$ and $P7_{seq}$	6.02	5.06	6.16	9.45	8.11	7.50
$P4_{seq}$ and $P7_{par}$	5.94	5.04	4.70	9.45	8.10	7.49
$P4_{seq}$ and $P7_{seq}$	6.57			10.61		

Table 1. Phases $P4$ and $P7$ - Execution time on a Power Challenge of different parallelization strategies and sequential execution time.

Parallelization strategies where chunks of iterations are cyclically assigned to processors lead to data distributions that can be modeled using the $CYCLIC$ or $BLOCK\_CYCLIC$ attribute in the HPF distribution directive.

2.2 Iteration Space Partitioning

In most commercial compilers, the iteration space of each parallel loop is partitioned in equally sized chunks trying to obtain a good load balancing. We propose to partition the iteration space of each loop in a sequence of phases so that each processor executes chunks of iterations that access the same array sections (when possible). This is useful to partition the iteration space when the bounds of the iteration space change from a phase to another, when offsets are used in array subscript expressions, or when we have a sequential loop that accesses distributed data. In all these cases, an adequate assignment of iterations to processors might reduce or eliminate data movements. The compiler has to insert code so that each processor computes its lower and upper bound of the iteration space that it has to execute. The owner computes rule is used to drive this iteration partitioning; this rule states that the owner of a piece of data is the responsible for its update along program execution. The ownership can change dynamically if considered profitable. We name this feature chunk affinity partitioning. The compiler has to detect if this partitioning lead to a loss of load balancing and decide an intermediate solution.

	NUM=64			**NUM=256**		
	P=2	**P=4**	**P=8**	**P=2**	**P=4**	**P=8**
$P4_{par}$ and $P7_{seq}$	6.02	5.06	6.16	9.45	8.11	7.50
$P4_{par}$ and $P7_{chunk}$	5.28	4.82	7.36	8.14	6.91	6.58

Table 2. Phases $P4$ and $P7$ - Execution time on a Power Challenge with static row distribution.

For instance, assume that we execute phases $P4$ and $P7$ in ADI with a static row distribution $(BLOCK, *)$. In this case, and due to data dependences, phase $P4$ can be executed fully in parallel and phase $P7$ must be executed sequentially. In most programming models, sequential phases are executed by a single processor; if so, additional data movement and invalidation overheads have to be paid in order to change the ownership of the data being computed and to bring to its cache all data needed to perform the sequential computation. To avoid these overheads, PDDT partitions the iteration space and inserts synchronization to preserve data dependences. In this way, each processor works with data it owns but it does not start execution until the previous processor completes its execution. The effects of this partitioning are shown in Table 2 and they reflect the trade off between synchronization and data movement; the first row shows the execution time when phase $P7$ is executed sequentially by a single processor; the second row shows the execution time when this phase is executed

in parallel ensuring that a processor does not start execution until the previous one has finished. The decrease in the execution time is due to the overhead of data movements and invalidations that are avoided[4], and it is more noticeable when the arrays that are moved are bigger.

2.3 Pipelined Computation

In this section we show how parallelization strategies can benefit from pipelined computations. In a pipelined computation, a processor cannot begin execution until its predecessor has partially finished its computation. For instance, this is useful to reduce the negative effects of data dependences. Although pipelined computations are well known, most of the currently available compilers do not apply them. In this section we also show how pipelined computations are useful to minimize overheads due to false sharing of cache lines.

In the previous section we have noticed the benefits of executing phase $P7$ in a synchronized way preserving data dependences. However, there exists room to improve the performance of the execution if we pipeline the execution of the loop. In this case, once a processor finishes the computation of a chunk of iterations of the j loop, the next processor can start its computation using data previously computed. The size of the chunk determines the amount of overlap and the overhead of synchronization incurred due to the pipelined execution. Figure 2 shows the execution time of this phase for different sizes of the chunk. Notice that with small chunks we are obtaining near optimal performance; this means that the synchronization overhead we are paying compensates the negative effect of the sequential execution. In addition, this model of computation also reduces the overheads due to data motion, since they are overlapped with computation.

We also propose to use pipelined computations to minimize the overheads introduced by false sharing. For ADI, false sharing appears when we use the data set with NUM=64 and P=8. In the Power Challenge, cache lines are 128 bytes long; therefore each cache line holds 16 elements of the arrays if they are double precision. So notice that when we distribute the arrays in a $(BLOCK, *)$ way, each cache line is shared by two processors. Figure 3.a shows the distribution of cache lines among processors. Therefore, additional movement and invalidation happen due to false sharing. The negative effect of false sharing can be observed in Table 1 for the two parallelizations strategies that execute phase $P4$ in parallel; notice that the execution time with P=8 is greater than with P=4. The same effects can be observed in the first plot in Figure 2, where one can see that the execution time of the phase has an anomalous behavior for P=8.

The main idea behind the pipelined execution is that a processor starts using a set of cache lines when another conflicting processor finishes using them. To reduce the overheads of false sharing, we propose to independently pipeline the computation of all the processors that share cache lines. This synchronized execution model also allows PDDT to perform an estimation of the additional

[4] The experiment with NUM=64 and P=8 shows a performance degradation due to the false sharing; this situation is considered in the next section.

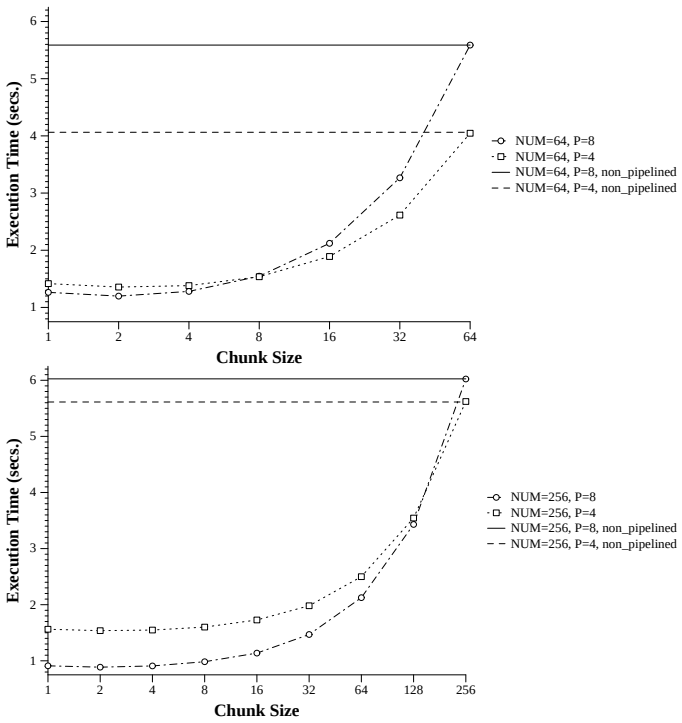


Fig. 2. Phase $P7$ - Using pipelined computations to minimize the negative effect of dependences on the execution time.

data movement that appears due to false sharing; if the execution is not synchronized, the additional costs become unpredictable at compile time. So for instance, in the previous case, PDDT would decide to pipeline the execution of processors 0 and 1, 2 and 3, and so on, as shown in Figure 3.b. Figure 4 shows the execution time of phase $P4$ for different sizes of the chunk. Again, notice that the overhead introduced by synchronization and the reduction of parallel execution in the pipelined model clearly compensates the negative effect of false sharing. For chunks smaller than 32, the pipelined execution improves over the fully parallel execution. However, for chunks bigger than 16 the sequentialization of the execution is worse than the negative effect of false sharing.

Table 3 shows the execution time of phases $P4$ and $P7$, when pipelining is used to minimize the negative effects of dependences and false sharing. In cases where false sharing does not occur (NUM=256 and P=8), pipelining reduces performance; the overheads introduced by synchronization and the loss of parallelism in the pipelined model are the causes of this degradation. However, when false sharing happens (for NUM=64 and P=8) these overheads compensate the additional data movement costs. Figure 5 shows the code generated by PDDT for phases $P4$ and $P7$; loops have been parallelized and pipelined (chunk size 4) for a static $(BLOCK, *)$ distribution.

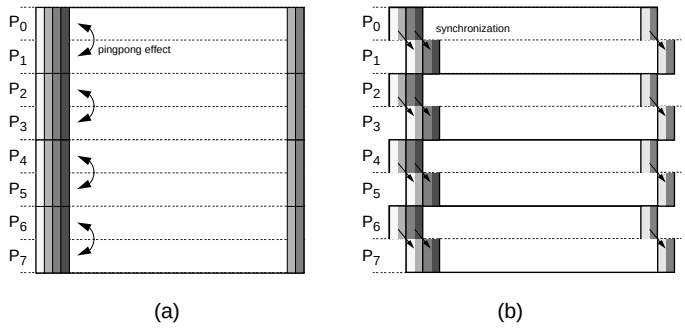


Fig. 3. Phase $P4$ - (a) False sharing of cache lines in phase $P4$ and (b) Pipelined execution to minimize its negative effects.

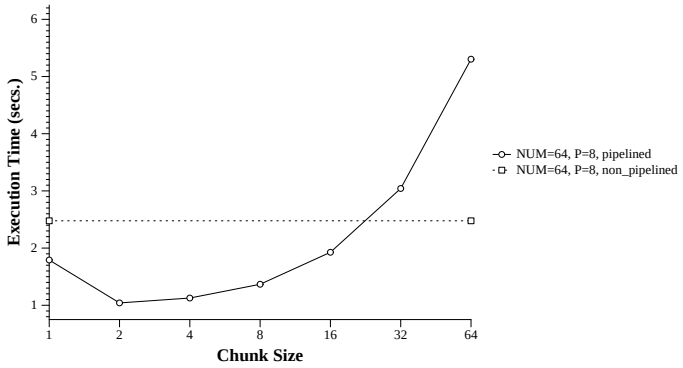


Fig. 4. Phase $P4$ - Using pipelined computations to minimize the negative effects of false sharing on the execution time.

In order to reduce false sharing in the access to synchronization objects, they are padded to the size of the cache line and aligned to cache line boundaries. If not done, several elements of the synchronizing object are located on the same cache line; when a processor writes to one of these elements, the other elements co-located on the same cache line are invalidated and new invalidation misses appear when the waiting processors re-read their status.

3 Parallelization and Data Distribution Process in PDDT

Our research tool (PDDT - Parallelization and Data Distribution Tool) analyzes Fortran77 programs and annotates them with directives and executable statements of shared memory (Convex Exemplar, SGI Power Fortran) programming models. The structure of loop nests may be changed in order to minimize data motion, improve locality of references and minimize false sharing. These decisions are done so that the amount of remote accesses is reduced as much as possible, while maximizing the parallelism achieved.

	NUM=64	NUM=256
$P4_{par}$ and $P7_{par}$	2.71	1.74
$P4_{pipe}$ and $P7_{par}$	1.61	1.93
$P4_{par}$ and $P7_{pipe}$	3.28	1.57
$P4_{pipe}$ and $P7_{pipe}$	2.25	1.75

Table 3. Phases $P4$ and $P7$ - Execution time on a Power Challenge (P=8) using pipelined computation in phase $P4$ to minimize false sharing and in phase $P7$ to minimize the effect of dependences (chunk size 2). Rows 1 and 2 correspond to the dynamic solution; rows 3 and 4 correspond to static $(BLOCK, *)$ distribution.

```
     do 10 iter = 1, MAXITER
          do jj = 1, NUM, 2
               token(jj) = NUM/2
               token(jj+1) = 0
          enddo
C$PAR PARALLEL DO LOCAL(i, j, jj, my$p, lb$i, ub$i, next$p)
          do my$p = 0, 7                                    Phase P4
               lb$i = max((my$p * NUM / 8) + 1, 1)
               ub$i = min((my$p + 1) * (NUM / 8), NUM)
               next$p = 1
               do jj = 2, NUM, 4
444                 if (next$p .gt. token(my$p+1)) goto 444
                    do 4 j = jj, min(jj + 3, NUM)
                         do 4 i = lb$i, ub$i
                              ...
4                   continue
                    token(my$p + 1) = token(my$p + 1) + 1
                    next$p = next$p + 1
               enddo
          enddo
          do jj = 1, NUM
               token(jj) = 0
          enddo
C$PAR PARALLEL DO LOCAL(i, j, jj, my$p, lb$i, ub$i)
          do my$p = 0, 7                                    Phase P7
               lb$i = max((my$p * NUM / 8) + 1, 2)
               ub$i = min((my$p + 1) * (NUM / 8), NUM)
               do jj = 1, NUM, 4
777                 if (token(jj) .ne. my$p) goto 777
                    do 7 j = jj, min(jj + 3, NUM)
                         do 7 i = lb$i, ub$i
                              ...
7                   continue
                    token(jj) = token(jj) + 1
               enddo
          enddo
10   continue
```

Fig. 5. Transformed code for phases $P4$ and $P7$ according to $(BLOCK, *)$ distribution; phase $P4$ has been parallelized with pipelining to minimize false sharing. Phase $P7$ has been parallelized for chunk affinity and pipelined to minimize the sequentilization due to dependences. Chunk size is 4 in both phases.

PDDT is targeted to generic Non-Uniform Memory Access Architectures (NUMA) with local and remote memory accesses. Each processor has its own memory hierarchy and can access the memories in other processors through the interconnection network. Data movement costs are estimated as the number of cache lines that need to be transferred multiplied by the remote access time. Given a parallelization strategy, computation costs are estimated from a profile of the sequential execution on a workstation based on the same processor and with the same memory hierarchy than the parallel machine (which is a common fact in most of the hardware vendor product lines).

All cost estimations in PDDT are done numerically assuming some problem and machine specific parameters. Profiling the sequential execution of the original Fortran 77 program is required in order to obtain these problem specific parameters, such as array sizes, the number of iterations for the loops and their execution time, and the probabilities of the different branches in conditional statements. A configuration file allows the user to specify some machine specific parameters (number of processors, overhead of parallel thread creation, local and remote memory access costs, ...), to restrict the kind of solutions explored by PDDT (number of distributed dimensions and loops to parallelize, static or dynamic solutions, number of candidate mappings for the phases and procedures, ...), and to specify the target programming model.

PDDT has evolved from our automatic data distribution tool (DDT) targeted to distributed-memory machines. Details about its implementation can be found elsewhere [AGG+94, AGG+95]. The main steps of the parallelization process performed by PDDT are outlined below:

- Detection of phases or computationally intensive portions of code, which mainly correspond to nested loops and calls to procedures. Phases are considered at this level as portions of code that modify the contents of the cache. The definition of phase by [KK95] is used.
- Selection of candidate solutions for the previously detected phases and estimation of their cost. Each solution represents a particular distribution of the elements of the arrays across the private caches and a parallelization strategy. For each phase, PDDT decides which loops to parallelize, which loops must be executed sequentially, and which ones benefit from pipelining. Detection of false sharing and minimization of its effects using pipelining are done at this stage of the parallelization process. The decisions are done based on an estimation of the computation and movement costs, which are both affected by the chunk size selected when the pipelined execution model is used. To do that, an analysis of the reference patterns and data dependences within the scope of phases is done.
- Analysis of compatibility among phases, and selection of solutions for each of them. This selection is done by exploring a search space composed of the different candidate solutions for each phase and estimating the data movement costs due to the remapping of arrays between phases. This analysis is done by characterizing cache contents in terms of HPF-like data mappings; in a cache-based system more than one mapping function may be needed to

characterize it after the execution of a phase. The phase control flow graph drives the process and identifies the different sequences of phases that might appear during the execution of a procedure.
- Code restructuring: generation of shared-memory parallelization directives that specify loops that are run sequentially or in parallel. In addition to that, PDDT also specifies the partitioning of the iteration space for the loops, changes in the structure of loop nests to improve spatial locality and introduces synchronization to guarantee the correct behavior of the program and to minimize false sharing. In this step, some changes in the declaration of data structures are also performed: (i) dummy arrays are inserted in order to guarantee that all major data structures are aligned on cache line boundaries; (ii) if the first dimension of an array is distributed, it is padded with "unused" elements in order to have an integral number of cache lines allocated to this dimension.

The process described above is done under control of the inter-procedural analysis module; this module builds the call graph for the entire program and records information about call sites and actual arguments. Once built, a bottom-up pass over the call graph decides the order in which procedures are analyzed, analyzes them and records information into the PDDT inter-procedural database.

The native compiler for the target machine is used to translate the annotated Fortran77 code generated by PDDT into an efficient code, taking care of all the aspects related to scalar optimizations, further locality exploitation and proper storage of the arrays.

4 Experimental Results

PDDT can be used either as a parallelizing tool or as a prediction tool able to help the user in writing parallel programs for cache-coherent shared-memory multiprocessors. In both cases, PDDT hides all the main architectural features of the target machine and guides the user during the parallelization process, showing him the sources of inefficiency. Given a (partial or complete) parallelization strategy for the program, PDDT estimates the cost of executing the program on the target machine, both in terms of computation and data motion costs. If the parallelization strategy is not complete, PDDT parallelizes those loops not specified by the user according to the user supplied parallelization for the rest of the loops.

In this section we analyze two programs: the Alternate Direction Implicit integration kernel *ADI*, and *swm* from the SPEC92 benchmark set. For the first one, we will show how PDDT generates parallelization strategies that are better than the ones generated by compilers that perform the parallelization process without caring about data distribution. In particular, we compare with the *pfa* compiler for the Power Challenge SGI architecture. For the second one, we will analyze the accuracy of the performance estimations performed by PDDT.

4.1 Alternate Direction Implicit ADI

The ADI kernel has a two-dimensional data space and has a set of computational phases that perform forward and backward sweeps along rows and columns of the data space. Three different array distribution strategies are evaluated (static row $(BLOCK, *)$, static column $(*, BLOCK)$ and dynamic in which some phases are executed with row distribution and some other phases with column distribution); for each of these strategies, different optimizations are turned on in order to evaluate their impact on performance (chunk affinity and pipelined computation to minimize dependences and false sharing). Table 4 shows the speed up obtained over the sequential execution on a single processor. The target architecture is a Silicon Graphics Power Challenge with eight R8000 processors, and 4 Mbyte of private cache per processor.

w/o Chunk Affinity and w/o Pipelined Execution						
	NUM=64			**NUM=256**		
	P=2	**P=4**	**P=8**	**P=2**	**P=4**	**P=8**
(BLOCK, *)	1.18	1.37	1.12	1.18	1.39	1.52
(*, BLOCK)	1.18	1.38	1.50	1.18	1.39	1.51
Dynamic	1.77	3.30	2.49	1.79	3.40	6.50
w/ Chunk Affinity and w/o Pipelined Execution						
	NUM=64			**NUM=256**		
	P=2	**P=4**	**P=8**	**P=2**	**P=4**	**P=8**
(BLOCK, *)	1.24	1.33	0.87	1.30	1.52	1.60
(*, BLOCK)	1.30	1.52	1.59	1.32	1.58	1.72
Dynamic	1.77	3.30	2.49	1.79	3.40	6.50
w/ Chunk Affinity and w/ Pipelined Execution						
	NUM=64			**NUM=256**		
	P=2	**P=4**	**P=8**	**P=2**	**P=4**	**P=8**
(BLOCK, *)	1.75	2.94	3.22	1.92	3.66	6.51
(*, BLOCK)	1.80	3.03	4.50	1.94	3.71	6.70
Dynamic	1.77	3.30	4.47	1.79	3.40	6.50

Table 4. ADI - Speed up for different program sizes and number of processors. Three data distributions are evaluated: static row, static column and dynamic distribution. Chunk affinity and pipelined computation are turned on and off to show their effect on performance.

The following conclusions can be drawn from the figures in Table 4. First of all one can see that in the static solutions it is important to partition the iteration space so that data are re-used by the same processor as much as possible; in particular for this code, to execute sequential phases using the processors that at the time are the owners of the distributed arrays. In general, the speed up of row distribution is worse than column distribution since the amount of data moved around in the first one is bigger (less elements in each cache line moved are useful). In addition, the row distribution suffers from false sharing, which even degrades more performance.

We can also observe that using pipelined computations to reduce the negative effect of dependences and false sharing reduces the performance gap between the dynamic and static solutions. Depending on the problem size and number of processors, static column distribution or dynamic distribution are automatically selected by PDDT. This is not the case for the native SGI compiler, which always selects the dynamic solution and does not perform pipelining to control false sharing.

4.2 Swm Benchmark

Finally we show the accuracy of PDDT in the performance prediction of the automatically parallelized programs. For this purpose we use one of the programs in the SPEC benchmark set. The program has been parallelized using PDDT configured with the architectural parameters of a Power Challenge Silicon Graphics multiprocessor system. The parallelized program, annotated with PFA directives, is fed into the native Fortran compiler.

Table 5 shows the actual and predicted speed-ups of the parallelized program using 2, 4, or 8 processors and two different problem sizes: 64 and 512. Notice that the accuracy of the prediction is enough to validate the data mappings and parallelization strategies suggested for this code.

	size=64		**size=512**	
Number of CPUs	**Predicted**	**Measured**	**Predicted**	**Measured**
2	1.9115	1.9126	1.9979	2.0246
4	3.6491	3.5064	3.9870	4.0289
8	6.6888	6.2407	7.3879	7.9543

Table 5. Predicted and measured speed-ups for swm on the Power Challenge SGI.

5 Conclusions and Future Work

PDDT is a flexible parallelizing compiler for cache-based shared-memory multiprocessors. It can automatically parallelize loops and change their structure based on tracking the dynamic placement of data along program execution. In these architectures a number of CPUs can simultaneously access data anywhere in the system. However, the non-uniformity of the memory accesses is an important issue to consider and may require a higher programming effort in order to achieve performance; trying to access those levels in the hierarchy closer to the processor may increase execution efficiency.

In this paper we have presented the set of features included in PDDT that most influence the selection of parallelization strategies for the loops in numerical programs: partitioning of the iteration space, and pipelined computation to minimize sequentialization and false sharing. The process relies on technology previously developed for automatic data distribution for distributed-memory systems. We have evaluated the quality of the solutions generated by PDDT by comparing the performance of the solutions suggested against the performance

of solutions generated by the native compiler of a SGI Power Challenge system. We have also shown how the predicted speed-ups are close to the actual ones obtained when then program is executed. PDDT handles partially annotated Fortran 77 programs with directives that specify parallelization strategies; in this case PDDT is useful as a support tool for the developer of parallel codes in estimating the effect of user selected parallelization strategies in the final performance of the parallel program.

In this paper we have assumed that conflict misses never happen. This is not true in real systems with finite caches. Although this has not a severe impact for the programs we have evaluated (because of the size of the data sets and the size of each private cache in the SGI Power Challenge - 4 Mbyte), this is a topic of current research. The same technology can be used to perform a software controlled data prefetching and preflushing between computational phases.

References

[AAL95] J.M. Anderson, S.P. Amarasinghe, and M.S. Lam. Data and computation transformations for multiprocessors. In *Principles and Practice of Parallel Programming*, pages 166–178. ACM SIGPLAN, June 1995.

[AGG+94] E. Ayguadé, J. Garcia, M. Gironès, J. Labarta, J. Torres, and M. Valero. Detecting and Using Affinity in an Automatic Data Distribution Tool. In K. Pingali et al., editor, *Proceedings of the 7th Annual Workshop on Languages and Compilers for Parallel Computing*, pages 61–75, Ithaca, NY, August 1994. Lecture Notes in Computer Science vol. 892, Springer-Verlag.

[AGG+95] E. Ayguadé, J. Garcia, M. Gironès, M.L. Grande, and J Labarta. Data Redistribution in an Automatic Data Distribution Tool. In C.-H. Huang et al., editor, *Proceedings of the 8th Annual Workshop on Languages and Compilers for Parallel Computing*, pages 407–421, Columbus, Ohio, August 1995. Lecture Notes in Computer Science vol. 1033, Springer -Verlag.

[AL93] J.M. Anderson and M.S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Conference on Programming Language Design and Implementation*, pages 112–125. ACM SIGPLAN, June 1993.

[BCG+95] P. Banerjee, J.A. Chandy, M. Gupta, E.W. Hodges IV, J.G. Holm, A. Lain, D.J. Palermo, S. Ramaswamy, and E. Su. The Paradigm Compiler for Distributed-Memory Multicomputers. *IEEE Computer*, 28(10):37–47, October October 1995.

[Con94] Convex. *SPP1000 Systems Overview*. Convex Computer Corporation, 1994.

[JE95] T.E. Jeremiassen and S.J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Principles and Practice of Parallel Programming*, pages 179–188. ACM, June 1995.

[KK95] K. Kennedy and U. Kremer. Automatic Data Layout for High Performance Fortran. In *Proceedings of Supercomputing'95, San Diego, CA*, December 1995.

[SGI96] Silicon Graphics Computer Systems SGI. *Power Challenge Technical Report*, 1996.

[SSGC95] T.J. Scheffler, R. Schreiber, J.R. Gilbert, and S. Chatterjee. Aligning Parallel Arrays to Reduce Communication. In *Frontiers95: The 5th Symposium*

on the Frontiers of Massively Parallel Computation, pages 324–331, February 1995.