Scheduling Non-Uniform Parallel Loops on Highly Parallel Multicomputers

Salvatore Orlando

Dept. of Appl. Math. and Comp. Science
Ca' Foscari Venice University
Venezia Mestre, 30173 Italy
orlando@unive.it

Raffaele Perego

Istituto CNUCE
National Research Council (CNR)
Pisa, 56126 Italy
r.perego@cnuce.cnr.it

Abstract

We consider the problem of scheduling parallel loops that are characterized by highly varying execution times (non-uniform parallel loops), and whose iterations operate on large data structures. A general parallel loop implementation template for message-passing multicomputers is presented. It exploits a partial static replication of the data and a two-phase scheduling strategy. While the first phase of the strategy is static, the second one is dynamic, and only starts when processor loads are estimated to be lower than a given threshold. The static knowledge of the data distribution strategy, which randomly replicates blocks on a fixed set of partner processors, is successfully exploited by the dynamic scheduling phase of our technique. The main advantages of the approach are low-cost scheduling decisions (mainly based on local information) and high scalability, since communications to request further work are overlapped with computations.

A simple analytical model of load imbalance, used to tune the template and approximate the actual performances, is also introduced. The template can be used by compilers to generate well-balanced schedules of loops when no static information on processor workloads can be derived. Results of experiments conducted on a 128-node nCUBE 2 are reported and discussed.

1 Introduction

Loops with independent iterations (*parallel loops*) are the largest source of parallelism in many time-consuming scientific applications. Restructuring compilers for sequential or data parallel programming languages successfully exploit this source of parallelism by distributing loop iterations among different processors of the target parallel machine. Loop schedule can be determined statically or dynamically. Dynamic scheduling may be needed when iterations are characterized by high variance in execution time, so that the adoption of a static schedule could cause an imbalance in processor workloads, thus reducing the potential speedup. Non-uniform loops are common in many application fields such as sparse matrix computations, image processing, Montecarlo calculations. Although the problem of finding an optimal schedule for non-uniform parallel loops is *NP*-hard [7], many effective dynamic scheduling heuristics for shared-memory multiprocessors have been proposed and experimented [10, 11, 8, 5, 2]. A less frequently studied problem is loop scheduling for distributed-memory architectures [7, 2], which involves strong relationships with other factors such as *data partitioning* and *locality of references*.

In this paper we consider distributed-memory machines that only supply mechanisms for message passing. Processors in these machines can only refer data that are allocated, either statically or dynamically, to the corresponding processor local memories.

Due to the large cost of dynamic policies, which may require a large amount of communications, usual implementations of parallel loops on multicomputers do exploit not only static allocation of data, but also static schedules. As an example, static schedules are exploited by compilers for data-parallel languages, such as HPF or FortranD [3, 4]. These languages require that programmers provide a data distribution scheme on an abstract architecture[1], while the compiler derives a static allocation of loop iterations on the basis of the specified distribution [1]. Since locality exploitation is the key to achieve speedup, the *owner computes rule* [13], which schedules computations to the owner of the corresponding data, is followed. This technique may incur in very little run time overheads if the loop iterations that have been

[1] Programmers can either specify regular data distributions, i.e. *block* or *cyclic*, or can specify irregular distributions by employing accurate knowledge about the computational load associated with the various parts of a given data structure.

scheduled to a given processor mainly refer to data stored in the local memory of the processor itself. Unfortunately, this static approach results in poor processor utilization when processor workloads turn out to be unbalanced.

On the other hand, pure dynamic scheduling approaches, which entirely build loop schedule at run time, balance the workload among processors, but result in high overheads and low scalability. The effective dynamic techniques that have been developed for shared-memory machines cannot be, in fact, simply adapted to distributed-memory environments. The single queue, which is used on shared-memory implementations to store the iteration indexes (*Self Scheduling* techniques [10, 11, 8, 5, 2]), should introduce a centralization point in the distributed implementation, thus jeopardizing scalability. Another problem is related to the allocation of data. In fact, to avoid data transfer overheads, it should be needed to replicate the full data set in the local memories of all the processors. Complete replication limits the applicability of the technique to problem instances whose data sets can be contained by the local memory associated with a single processor.

To overcome these shortcomings, we propose an implementation template for parallel loop execution that exploits a very simple distributed implementation of the global queue, and does not require accurate static knowledge about the way the computational load will be distributed on the data structures accessed. Data are partitioned and distributed, eventually taking advantage of the data distribution directives of data parallel languages like HPF. The static phase of the technique schedules the loop iterations according to the *owner computes rule*. When load imbalances are detected, a dynamic scheduling technique is exploited, based on the static knowledge of a small set of partners associated with each processor. The partners are randomly chosen so that, with respect to the initial data partition assignment, they are evenly scattered on the data structures. At run-time a processor *prefetches* further iterations from its partners only if its load is becoming lower than a given *threshold*. An accurate choice of the *threshold* strongly limits overheads by overlapping communications and computations. To further reduce communication overheads, partial data replications are introduced: the data partitions assigned to the partners of a generic processor p, are also replicated on p. This allows us to adopt a not aggressive prefetching policy which can be successfully exploited also for fine grain parallel loops. We have verified that the proposed strategy can solve very different load imbalance occurrences, and thus furnishes a general solution to the cases in which either load imbalance cannot be predicted at compile time, or its determination is too much expensive with respect to the achievable benefits.

The paper is organized as follows. Section 2 shortly surveys work related to the parallel loop scheduling problem. A simple analytical model of loop load imbalance is presented in Section 3, while Section 4 describes the proposed implementation template. Results of experiments conducted on a 128-node nCUBE 2 multicomputer are reported and discussed in Section 5. Conclusions follow.

2 Related work

The parallel loop scheduling problem has been investigated in depth by researchers working on shared-memory multiprocessors. Most proposals address the efficient implementation of loops by defining *Self Scheduling* policies which reduce synchronizations among processors by enlarging parallel task granularity. The main goal of these works is to determine the optimal number of iterations fetched by each processor at each access to the central queue (*chunk size*). Clearly, the larger the chunk size is, the lower the contention overheads for accessing the shared queue, and the higher the probability of introducing load imbalances.

Polychronopoulos and Kuck proposed *Guided Self Scheduling*, according to which $\frac{u}{P}$ iterations, where u is the number of remaining unscheduled iterations and P is the number of processors involved, are fetched at each time by an idle processor [10]. *Trapezoid Self Scheduling* [11] was proposed by Tzen and Ni to reduce the number of synchronizations by linearly decreasing the chunk size. Hummel, Schonberg and Flynn presented *Factoring* [5], which requires that P consecutive chunks of size k, where $k \leq \frac{u}{2 \cdot P}$, are inserted into the shared queue when it becomes empty.

Due to improvements in processor architectures with the exploitation of fine grain parallelism, processors are getting faster at a higher rate than memories and interconnection networks are. To overcome this problem, shared-memory multiprocessors are being equipped with even larger caches. This architectural trend is moving shared-memory multiprocessor even closer to distributed-memory counterparts. Thus, in both shared and distributed-memory machines, exploiting locality is recognized as one of the main requirements to achieve scalability [6]. The allocation of data in either local memories or caches of each processor must therefore be accurately considered to obtain effective scheduling algorithms.

As far as regards shared-memory environments, Markatos and LeBlanc [8] investigated locality and data reuse to obtain scalable and efficient implementations of non-uniform parallel loops on shared-memory multiprocessors. They explored a scheduling strategy, based on a static partitioning of iterations, which ini-

tially assigns iterations for *affinity* with previously assigned ones. Affinity regards the presence of accessed data in the processor caches. The dynamic part of the technique, which performs run-time load balancing, is postponed until a load imbalance occurs. This approach is similar to ours, though we explicitly have to take into account data allocation/replication in the local memories of each processor.

Liu and Saletore worked on Self Scheduling techniques for distributed-memory machines [7]. They attempted to overcome the shortcomings mentioned in Section 1 by providing a hierarchical and distributed implementation of the centralized manager, and by investigating partial replication techniques to increase problem sizes.

3 A model of load imbalance

In this paper we consider loops which access arrays, though similar techniques could be devised for other types of data structures. Suppose that we have a parallel loop that reads an array and produces a *new* array with the same dimensions. Let D be the number of iterations in the loop, where each independent iteration produces a new element in the output array. If the sequential time to compute all the D iterations is T, the average time to compute each iteration is $\mu = \frac{T}{D}$. We have a non-uniform parallel loop when the iteration execution time may significantly differ from μ.

If the unbalanced iterations are evenly distributed on all the data structure, the problem of load imbalance can often be solved by adopting a simple coarse-grained *block* distribution along with a static schedule policy. In this case the probability of obtaining partitions with even loads is quite high.

However, we are interested in non-uniform parallel loops whose computational load is concentrated on one or more *regions* (i.e. subsets of contiguous elements) of the arrays. In this case, *block* distributions fail to evenly distribute the computational load.

Let $I = \{I_1, \ldots, I_D\}$ be the iteration space of the parallel loop, and $e(I_i)$ the execution time for iteration I_i. If the loop is non-uniform, there exists a subset $\overline{I} = \{I_{i_1}, \ldots, I_{i_k}\}$ such that $e(I_{i_j}) > \mu$ for all $j = 1, \ldots, k$. Thus $d = \frac{k}{D}$, $0 < d < 1$, represents the fraction of these more expensive iterations. Let H, where $H > \mu \cdot k$, be the total execution time required to compute the iterations in $\overline{I}$. Thus, $t = \frac{H}{T}$ is the fraction of total execution time spent in $\overline{I}$. From the above definitions we have:

$$1 > t = \frac{H}{T} > \frac{\mu \cdot k}{\mu \cdot D} = d > 0. \qquad (1)$$

The workload imbalance is clearly proportional to t. In fact, a high value of t means that the cost H is prevalent on T. The imbalance becomes even more difficult to treat when the load is too concentrated on a small portion of the data set. Thus, the imbalance is also inversely proportional to d.
From the above remarks, we can derive F, called *factor of imbalance*, as follows:

$$F = \frac{t}{d}. \qquad (2)$$

The value of F is directly proportional to load imbalance. The more the value of F approaches 1, the more the loop is uniform. On the other hand, high values of F (obtained for $d \to 0$ and $t \to 1$) correspond to very unbalanced loops in which the load is very much concentrated on a small portion of the dataset.

If we adopt a static scheduling scheme by uniformly distributing the array with a *blocking* strategy and by assigning loop iterations according to the *owner computes rule*, the value of F can be used to derive the *worst-case* total execution time. Suppose that the processors involved are P, so that a block partition corresponding to $\frac{D}{P}$ iterations is assigned to each processors. Moreover, suppose that the portion of the dataset $(d \cdot D)$ over which the predominant part of the total computational load $(H = t \cdot T)$ is concentrated, is a region large enough to entirely contain almost one partition of the dataset. Let $\overline{B}$ be the most loaded partition and $\overline{p}$ the processor holding it. Thus, to execute the parallel loop, processor $\overline{p}$ will take a time $T_{\overline{p}}$ approximately equal to

$$T_{\overline{p}} = \frac{t \cdot T}{d \cdot P} = F \cdot \frac{T}{P} = F \cdot \frac{\mu \cdot D}{P}. \qquad (3)$$

Note that, since $T_{\overline{p}}$ is the completion time of the most loaded processor, this time will also be the completion time of the parallel run of the loop.

4 A two-phase scheduling technique

Our hybrid scheduling technique relies upon a static data partitioning, and a distribution strategy that allocates these partitions to the various processors with partial *replications*. Loop iterations are statically scheduled until load imbalances are detected. The replicated partitions are exploited by the dynamic phase of the scheduling technique, which does not involve all the processors, but only the ones that are the owners of data partitions that are also locally replicated.

4.1 Data distribution strategy

In distributed-memory environments, a given loop iteration can be scheduled to a processor only if a copy

of the data that is needed to compute the iteration is stored in the processor local memory. Regarding the allocation of the data, we propose that:

1. data structures, if necessary *aligned* to an abstract topology, are partitioned into blocks of equal size. The number of blocks is equal to the number P of processors. For each block B_i, where $i \in \{0, \ldots, P-1\}$, a single processor p_i is chosen to be the *owner* of the block. B_i is called *primary partition* of p_i;

2. besides the above primary allocation, each block is also copied as a *secondary partition* to the local memories of *exactly* $m-1$, $m < P$, *distinct* processors. m is called *replication degree*;

3. each processor p holds the primary partition B_i and $m-1$ *distinct* secondary partitions $\{B_{i_1}, \ldots, B_{i_{(m-1)}}\}$. The owners of these secondary partitions are the *partners* of p.

The replication degree m clearly influences the outcomes of our technique, and may need to be adjusted to the number of processors (partitions) used, and to the factor of imbalance F. Note that, if p_i is the owner of B_i, and B_i is replicated on other $m-1$ processors, these processors are the only ones which may require p_i to migrate some loop iterations operating on B_i. Correspondingly, p_i can ask for further work only its m partners, i.e. the processors which are the owners of its secondary partitions.

Since we are assuming that the load is very concentrated on some unknown regions of the data set, the m distinct partitions stored in each processing node must be randomly chosen so that they are evenly distributed on the data structure.

4.2 Scheduling strategy

The scheduling technique exploits the static knowledge about the allocation of primary and secondary copies of each block. The first part of the technique resembles a static schedule, according to which each processor performs iterations which update its primary partition. The second part is dynamic, as it only takes place if a load imbalance occurs. The interactions which occur during the dynamic phase are limited to the processors storing the same blocks, and an effective prefetching mechanism is implemented to overlap communications with computations. Furthermore, when the workloads deriving from the static allocation of the primary partitions are well balanced, the dynamic phase is never started. In this case, the run-time overhead becomes comparable to the one that derives from the adoption of a static scheduling scheme.

```
Schedule(queue Q, QR; process prtn_list[m-1])
{
  chunk T;
  while (!Terminated(prtn_list,Q,QR)) {
     if (Local_Load(Q,QR) < THRESHOLD)
        Prefetch_Chunks(prtn_list);
     T = Get_Chunk(Q,QR);
     Execute_Chunk(T);
     Handle_Msg(prtn_list,Q,QR);
     }
 }
```

Figure 1: Pseudo-code of the scheduling algorithm.

Figure 1 reports the pseudo-code of the scheduling algorithm executed by each processor p holding a primary partition B and $(m-1)$ secondary partitions. The iterations operating on the primary block of each processor are organized into chunks (i.e. batches of iterations identified by lower and upper limits), and inserted into a local queue `Q`. Each processor also manages a queue `QR`, used to store prefetched "remote" chunks operating on the $(m-1)$ secondary partitions. During the static phase of the technique queue `QR` is empty.

Function `Terminated(prtn_list,Q,QR)` returns `TRUE` *iff* queues `Q` and `QR` are empty, and all the partners of p (whose names are stored in the array `prtn_list`) have already sent a message to signal that their local loads do not allow them to give work away. This message is sent when the number of chunks stored in local queue `Q` is lower than a given `THRESHOLD`. The partners of p that have not yet sent this message are called the *active* partners of p.
Function `Local_Load(Q,QR)` returns an estimate of the total processor load, i.e. the number of chunks currently stored in the queues.
Subroutine `Prefetch_Chunks(prtn_list)`, which is executed when the total processor load is lower than a given `THRESHOLD`, sends to an active partner of p a message requiring a chunk to be inserted into `QR`. The active partner to which the request is sent is chosen on a *round-robin* basis. We found that it is more important not to waste time in making the choice than always making the best choice. Note that, in order to make the best choice, we should introduce further overheads within the scheduling strategy, since processors have to exchange information to maintain a global knowledge about the local load of all the partners involved.
Function `Get_Chunk(Q,QR)` first inspects queue `Q` and then queue `QR`. It gets a chunk from the first non empty queue; the first time the function discovers that `Q` is becoming empty, it notifies this event to the processors holding B as a secondary partition to

prevent further requests of chunks. The extracted chunk, if any, is soon processed within subroutine `Execute_Chunk(T)`.
Subroutine `Handle_Msg(prtn_list,Q,QR)` receives and manages messages that have arrived at processor p from its partners. The messages from partner $\overline{p}$ may contain:

1. a chunk that has been previously requested by subroutine `Prefetch_Chunks(prtn_list)`. The chunk is immediately inserted into queue `QR`;
2. a signal (sent within function `Get_Chunk(Q,QR)`), stating that, on the basis of its estimated load, $\overline{p}$ decided not to give away any more chunks. A flag is set in the $\overline{p}$ field of structure `prtn_list`;
3. the request to move a chunk from queue `Q` of p to queue `QR` of processor $\overline{p}$. The request is granted only if p estimates a load greater than `THRESHOLD`.

In principle, any well-known scheme (such as factoring [5], trapezoid self scheduling [11], guided self scheduling [10], etc.) can clearly be used to determine the sizes of chunks inserted into the local queue[2]. However, if the template is implemented on target parallel machines which do not provide efficient mechanisms to handle asynchronously arrived messages, the algorithm must periodically check the communication buffer for the presence of requests to be handled. This sort of polling mechanism can be simply implemented between the execution of two consecutive chunks (as in the pseudocode reported in Figure 1). It is more difficult to devise a general implementation in which the execution of a chunk is periodically interrupted for performing message handling. Thus, by scheduling chunks with decreasing sizes, a processor working on a large chunk may delay the handling of requests to migrate loop iterations. Such delays increase the overheads and affect the capability of unloaded processors to balance the load. These strategies could, on the other hand, be efficiently exploited by implementations of the template on parallel machines which supply efficient multitasking capabilities, or active messages [12], or high level interrupt handling. Without such mechanisms, uniformly small chunks must be preferred.

5 Experiments and results

We conducted many experiments to tune and evaluate our parallel loop implementation template. All the tests discussed here were performed on a 128-node nCUBE 2 multicomputer keeping μ and the sizes of the array partitions (100×100) fixed, while varying the number of processing nodes used, the factor of imbalance F, and the replication degree m. Thus, when the number of nodes used grows, the size of the problem increases accordingly and the same per-processor average load ($100 \cdot 100 \cdot \mu$) is maintained. In most experiments we used synthetic loads from a dummy non-uniform loop whose iterations update a bidimensional array, and have a cost proportional to the value of the array elements processed. By providing synthetic loads we could verify the behavior of the scheduling algorithm for many different problem instances.

[2] If chunk sizes are non-uniform, chunks have to be inserted into `Q` according to their size, and in descending order. This means that, at the beginning, potentially larger chunks are scheduled.

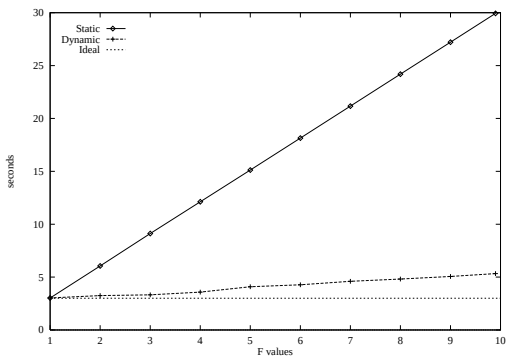

Figure 2: Completion times on 64 processing nodes as a function of the *factor of imbalance.*

In Figure 2 the completion time curves on 64 processing nodes as a function of the *factor of imbalance* F are plotted; one curve relates to our scheduling method with a replication degree 8 (curve labeled `Dynamic`), while the other to static scheduling (curve labeled `Static`). These tests were carried out by keeping the average per iteration execution time fixed ($\mu = 0.3$ msec.) so that a curve $y = k$, with constant k equal to 3 seconds ($100 \cdot 100 \cdot \mu$), represents the ideal completion time (curve labeled `Ideal`) independently of the value of F. As can be seen, substantial performance improvements over static scheduling were achieved. For $F = 1$ we obtained about the same times, for $F = 5$ our scheduling algorithm achieved a 3.7 speedup over static scheduling, while, for $F = 10$ the speedup increased to 5.6. Moreover, the curve of our implementation template keeps close enough to the optimal curve. Note that the static scheduling curve grows very regularly and, as estimated in section 3, has a slope approximately expressed by Equation (3): $\frac{\mu \cdot D}{P} = \frac{(0.3 \cdot 10^{-3}) \cdot 800^2}{64} = 3$.

Similar remarks can be made for the curves in Figure 3. This figure compares, for different *factors of*

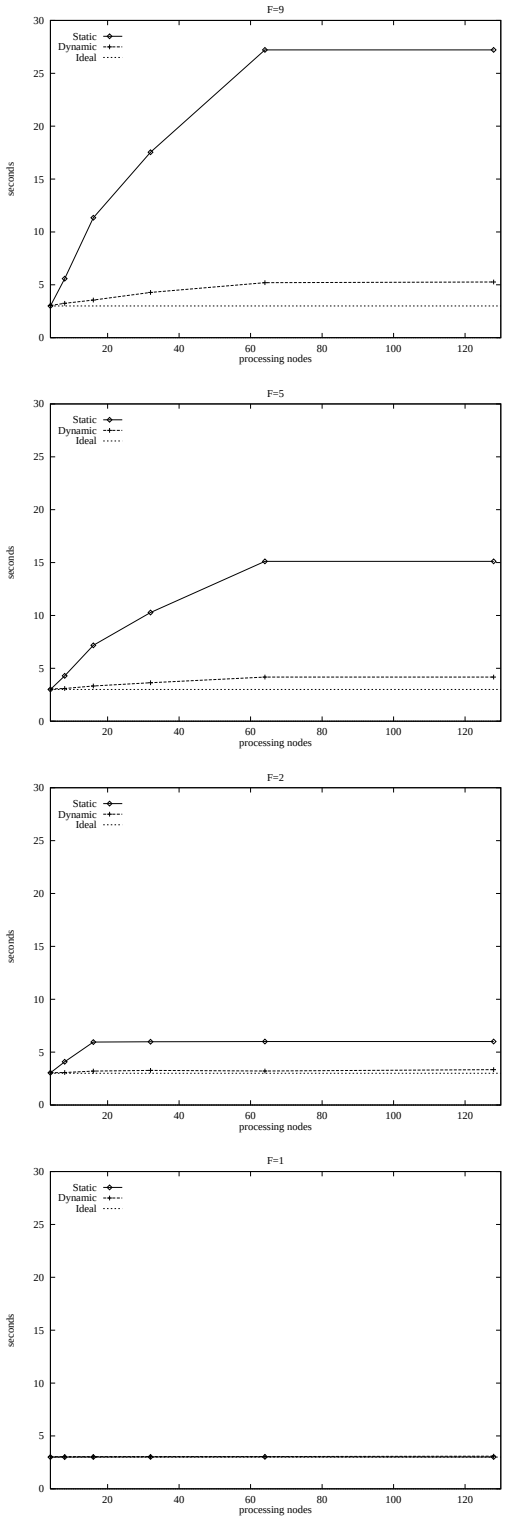

Figure 3: Comparison between static scheduling and our approach for different *factors of imbalance* as a function of the number of nodes used.

imbalance, static and dynamic scheduling as a function of the number of nodes used. The differences are large for higher values of F, and become less and less evident as the *factor of imbalance* decreases to 1. Note that the `Dynamic` curve obtained by executing a uniform loop (i.e. $F = 1$) exactly overlaps the `Static` curve, thus indicating that overheads are in this case negligible. Moreover, all `dynamic` curves have a slope that approaches 0, thus showing the good scalability of the implementation template.

The chunk size was chosen enough small to avoid, as previously said, large delays in handling requests for load migration. The size depends on μ, i.e. the average time to execute an iteration, and on T_{Comm}, i.e. the time to require and receive a new chunk from an active partner. μ and T_{Comm} depend on the specific problem instance, and on the cost of the machine mechanisms. More specifically, the chunk size was chosen so that its "average" execution time is approximately equal to T_{Comm}. On the basis of the chunk size, we devised [9] an analytical model that allows us to determine a value of `THRESHOLD` that makes effective the prefetching policy adopted. Note that, while large values of `THRESHOLD` produces unuseful chunk prefetching and may leave some processors unloaded, small values of `THRESHOLD` may block the activity of some processors that are waiting for remote chunks. The last parameter of the template that must be tuned is the replication degree. Experimentally we found that, for values of F from 1 to 10, a good choice of the replication degree is the square root of the number of processors used.

6 Conclusions and future work

A template for the implementation of a non-uniform parallel loop on distributed-memory machines has been presented and evaluated. The template works very well for loops that operate on arrays, and whose computational load is concentrated on one or more regions (i.e. subsets of contiguous elements) of the arrays. We presented an elegant model for this type of load imbalance.

The implementation technique is based on a static partitioning of the problem data, which are allocated on the various processing nodes with a fixed degree of *replication*. In this way, a given iteration can be executed on all the processors which store a copy of the data needed. The implementation template follows a static scheduling policy until a load imbalance occurs. At this point, the partial data replication allows some run-time scheduling decisions to be made, aimed at balancing processor workloads. The technique scales very well since small overheads are introduced: in fact, the set of partners to which each processor can ask for further work is statically fixed and

is very small, and communications and computations are overlapped by adopting an accurate prefetching policy. On the other hand, if the processor workloads are well balanced, the dynamic phase is never started and the run-time overhead of our template becomes about equal to the one derived from the adoption of a static scheduling scheme.

It is worth considering that data replication has been introduced to reduce the cost of communications needed to prefetch work from other processor. However, we discovered that one of the main advantages of this technique is the static limitation of the set of partners to which each processor can ask for further loop iterations. Therefore, when communication costs mainly depend on startups and only partially on message sizes, the same technique could be successfully adopted without replicating data.
On the other hand, if replication is adopted and a parallel loop must be iterated several times, i.e. is nested within an outer loop L, we have the problem of the coherence of all the replicated data partitions, even though the exploited replication degrees is very small. The coherence problem can be solved by sending the updated array elements to the owner of the corresponding partitions (this communication is not necessary if the new element has been produced by the owner of the data partition), and to all the processors that hold a copy of the data partition (one of these communications is not necessary if, as a consequence of a dynamic migration, the new element has been produced by one of these processors). From some preliminary experiments, we found that all these coherence communications do not introduce large overheads as we can overlap the communications relative to the iteration i of the outer loop L, with the static scheduling phase relative to the iteration $i+1$ of the loop.

Several results of experiments conducted on a 128-node nCUBE 2 multicomputer have been presented and discussed. The substantial performance improvements over static scheduling were proportional to the factor of imbalance F. For highly unbalanced loops, our template resulted in speedups of up to 5.6 by adopting a replication degree 8 on 64 processing nodes.

References

[1] B. Chapman, P. Mehrotra, H. Moritsch, and H. Zima. Dynamic Data Distribution in Vienna Fortran. In *Proc. of Supercomputing '93*, pages 284–293, 1993.

[2] H. El-Rewini, T.G. Lewis, and H.H. Ali. *Task Scheduling in Parallel and Distributed Systems.* Prentice Hall, 1994.

[3] High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993. Version 1.0.

[4] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM*, 35(8):67–80, August 1992.

[5] S.F. Hummel, E. Schonberg, and L.E. Flynn. Factoring: A Method for Scheduling Parallel Loops. *Communications of the ACM*, 35(8):90–101, August 1992.

[6] K.L Johnson. The Impact of Communication Locality on Large-Scale Multiprocessor Performance. In *Proc. of 19th Int. Symp. on Computer Architecture*, pages 392–402, 1992.

[7] J. Liu and V. A. Saletore. Self-Scheduling on Distributed-Memory Machines. In *Proc. of Supercomputing '93*, pages 814–823, 1993.

[8] E.P. Markatos and T.J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994.

[9] S. Orlando and R. Perego. Exploiting Partial Replication in Unbalanced Parallel Loop Scheduling on Multicomputers. Technical report, Ca' Foscari University, Venice, TR 03-95, 1995 (Submitted for pubblication).

[10] C. Polychronopoulos and D.J. Kuck. Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, 36(12), December 1987.

[11] T.H. Tzen and L.M. Ni. Dynamic Loop Scheduling on Shared-Memory Multiprocessors. In *Proc. of Int. Conf. on Parallel Processing - Vol II*, pages 247–250, 1991.

[12] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. Technical report, University of California, Berkeley, UCB/CDS 92/675, 1992.

[13] H.S. Zima and B.M. Chapman. Compiling for Distributed-Memory Systems. *Proceeding of the IEEE*, pages 264–287, February 1993.