Exploiting Partial Replication in Unbalanced Parallel Loop Scheduling on Multicomputers

Salvatore Orlando

Dept. of Applied Math. and Comp. Science, Ca' Foscari Univ., via Torino, 155 - 30173 Venezia Mestre, Italy – E-mail: orlando@unive.it

Raffaele Perego

CNUCE - C.N.R., via S. Maria, 36 - 56126 Pisa, Italy – E-mail: r.perego@cnuce.cnr.it

Abstract

We consider the problem of scheduling parallel loops whose iterations operate on large array data structures and are characterized by highly varying execution times (*unbalanced* or *non-uniform* parallel loops). A general parallel loop implementation template for message-passing distributed-memory multiprocessors (*multicomputers*) is presented. Assuming that it is impossible to statically determine the distribution of the computational load on the data accessed, the template exploits a hybrid scheduling strategy. The data are partially replicated on the processor's local memories and iterations are statically scheduled until first load imbalances are detected. At this point an effective dynamic scheduling technique is adopted to move iterations among nodes holding the same data. Most of the communications needed to implement dynamic load balancing are overlapped with computations, as a very effective prefetching policy is adopted. The template scales very well, since knowing where data are replicated makes it possible to balance the load without introducing high overheads.

In the paper a formal characterization of load imbalance related to a generic problem instance is also proposed. This characterization is used to derive an analytical cost model for the template, and in particular, to tune those parameters of the template that depend on the costs related to the specific features of the target machine and the specific problem.

The template and the related cost model are validated by experiments conducted on a 128-node nCUBE 2, whose results are reported and discussed.

Keywords: Parallel loop scheduling. Load balancing. Implementation template.

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 iterations of parallel loops among different processors of the target parallel machine. The assignment of iterations to processors (*schedule*) can be determined statically or dynamically. Dynamic scheduling techniques may be needed to smooth the unbalanced and unpredictable processor workloads which derive from loops whose iterations are characterized by high execution time variance. These non-uniform loops are common in many application fields such as sparse matrix computations, image processing, and Montecarlo calculations.

Although the problem of finding an optimal schedule for non-uniform parallel loops is *NP*-hard [10], many effective dynamic scheduling heuristics for shared-memory multiprocessors have been proposed and tested [12,14,11,7,4]. A less frequently studied problem is loop scheduling for distributed-memory architectures [10,4], which involves strong relationships with other important issues such as *data partitioning* and *locality of references*.

On the other hand, the general problem of load balancing for these architectures has been studied in depth, and many dynamic strategies have been proposed [16,9]. In this paper, the load balancing problem is addressed by restricting the form of parallelism exploited to parallel loops. As shown in [13], the restriction of the computational model often allows much more effective implementations to be devised by applying the parallel programming methodology known as template-based [2] or skeleton-based [3].

We consider distributed-memory machines that only supply mechanisms for message passing (*multicomputers*). Processors in these machines can only refer data that are allocated, either statically or dynamically, to the corresponding processor local memories.

Implementations of parallel loops on multicomputers usually exploit not only static allocation of data, but also static schedules. For example, static schedules are exploited by compilers of data-parallel languages, such as HPF or FortranD [5,6]. These languages require programmers to 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 [1,17]. Such static approaches may result in poor processor utilization when processor loads cannot be predicted statically.

Conversely, pure dynamic strategies which entirely build the loop schedule at run time may be adopted. These strategies are directly derived from the

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

Self Scheduling techniques developed for shared memory environments [12]. A centralized queue is used by a master processor to store the iteration indexes and to manage the scheduling process. As a worker processor ends its work, it asks the master for a new batch of iterations. The centralized management of the queue is one of the main drawbacks of pure dynamic scheduling strategies. Several techniques can be exploited to reduce the overheads caused by this bottleneck. For example, the introduction of a hierarchy of balancing domains, each responsible for a subset of the queue, has been proposed [10,16]. Another problem is data allocation. If no restriction is imposed on the dynamic strategy, to avoid data transfer overheads, the full data set must be replicated in the local memories of all the processors. Complete replication limits the applicability of these techniques to problem instances whose data sets can be contained by the local memory of a single processing node.
This paper shows that a very simple distributed implementation of the global queue can be effectively exploited by a hybrid scheduling strategy. Both the queue which contains the iteration indexes, and the data accessed within the loop, are statically partitioned and allocated to the various processing nodes. Each data partition is also replicated on the local memories of a small number of other nodes. Iterations are statically scheduled until first load imbalances are detected. At this point an effective dynamic scheduling technique is adopted to move iterations among partner nodes holding the same data. The load balancing choices are made on the basis of very limited knowledge and only require a few interactions because the subset of processing nodes which hold the data accessed by each iteration is small and statically known. Moreover, a very effective prefetching policy allows most communications to overlap with computations.
We have verified, both theoretically and experimentally, that our technique can solve very different load imbalances, and thus provides a general solution for cases in which either load imbalance cannot be predicted at compile time, or its determination is too expensive with respect to benefits. The implementation template we have devised scales very well, as its performance approaches the ideal case, i.e. the maximum speedup allowed by using a given number of processing nodes. This behavior has been verified for different load imbalances, which have been synthetically generated on the basis of a general model of load imbalance. The template's high performance is mainly due to low scheduling overheads, and the iteration prefetching policy that hides most iteration migration delays.

The paper is organized as follows. Section 2 briefly surveys work related to the dynamic scheduling of parallel computations. A simple model of loop load imbalance is presented in Section 3, while Section 4 describes the proposed implementation template, and also presents the analytical model used to tune some parameters of the template. The results of experiments, conducted on a 128-node nCUBE 2 multicomputer, are reported and discussed in Section 5. Conclusions follow.

2 Related works

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 [12]. *Trapezoid Self Scheduling* [14] was proposed by Tzen and Ni to reduce the number of synchronizations by linearly decreasing the chunk size. Hummel, Schonberg and Flynn presented *Factoring* [7], 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 [8]. 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 [11] 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 initially 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 node.

Liu and Saletore worked on Self Scheduling techniques for distributed-memory machines [10]. 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.

Willebeek-LeMair and Reeves [16] presented several general load balancing strategies for multicomputers. They introduced a very interesting framework to classify the various strategies. The main items that characterize their framework are:

(i) **Processor Load Evaluation**, how each processing node estimates its own load if needed;
(ii) **Load Balancing Profitability Determination**, how a node can decide whether it is profitable or not to perform load balancing by taking into account the related overheads;
(iii) **Task Migration Strategy**, how the source and the destination of a task migration are determined.

They also presented a technique called *Receive Initiated Diffusion*, which, like ours, employs task prefetching when the local load is below a given threshold. Their technique, however, needs to maintain global knowledge of the load either on all the nodes involved, or on a subset of the nodes called *domain*.
Another interesting work that presents general load balancing techniques is the one by Kumar et al. [9]. Among others, they introduced a technique that adopts a *global round-robin* policy to select the processing node to which a further task must be requested. The technique does not assume any knowledge of the load, and thus it might not be very accurate in scheduling decisions. On the other hand, the technique does not waste any time to evaluate the best load balancing choices. Kumar et al. showed that the technique is actually very scalable, provided that an efficient contention-free implementation of the *global round-robin* is adopted. We agree with their conclusions and we used a similar technique to implement the *Task Migration Strategy*. However, our strategy is local rather than global since whenever a node needs further work, it sends a request to one of its partners chosen on a simple inexpensive round-robin basis.

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 target 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 can often be solved by adopting a simple coarse-grained *block* dis-

tribution. 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 input arrays. In this case, *block* distributions fail to evenly distribute the computational load, while *cyclic* distributions may cause loss of locality.
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:

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

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:

$$F = \frac{t}{d}. \tag{2}$$

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* [17], the value of F can be used to derive the *worst-case* total execution time. Suppose that the processing nodes involved are P, so that a block partition corresponding to $\frac{D}{P}$ iterations is assigned to each node. 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 processing node holding it. Thus, to execute the parallel loop, node $\overline{p}$ will take a time $T_{\overline{p}}$ approximately equal to

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

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

4 The scheduling strategy

Our goal is to find an implementation template which results in a good trade-off between static and dynamic strategies. Our technique relies upon a static data partitioning, and a distribution strategy that allocates these partitions to the processing nodes with partial *replications*. These replications are taken into account by a dynamic scheduling technique, which does not involve all the processing nodes, thus limiting overheads and preserving locality exploitation.

4.1 Data distribution strategy

In distributed-memory environments, a given loop iteration can be scheduled to a processing node only if that node holds a copy of the data that is needed to compute the iteration. Regarding the allocation of the data, we propose that:

(i) 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 processing nodes. For each block B_i, where $i \in \{0, \ldots, P-1\}$, a single node p_i is chosen to be the *owner* of the block. B_i is called *primary partition* of node p_i;

(ii) besides the above primary allocation, each block is also copied as a *secondary partition* on the local memories of *exactly* $m-1$, $m < P$, *distinct* processing nodes;

Consequently, each processing node p holds the primary partition B_i and $m-1$ *distinct* secondary partitions $\{B_{i_1}, \ldots, B_{i_{(m-1)}}\}$. The value of m is named *replication degree*. The degree of replication 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, however, that we are considering problems in which:

- a total replication of the data may not be proposed, due to the large size of the data sets involved;
- the run-time transfer of the data is not appropriate because the time needed to compute each iteration is lower than the time required to transfer the accessed data over the interconnection network.

The $m-1$ nodes on which each partition B_i is replicated, are the only nodes which may require node p_i to migrate some loop iterations operating on B_i. Correspondingly, each node p_j can ask for further work only the nodes which own its secondary partitions.
Since we are assuming that the load is very concentrated on some unknown regions of the data set, to increase the probability that an unloaded node will find a more loaded partner when looking for further work, the m distinct partitions stored in each node have to be chosen so that they are evenly distributed on the data structure.
Taking into account these requirements, we can reformulate our distribution policy as follows. Given the blocks $\{B_0, \ldots, B_i, \ldots, B_{P-1}\}$ which must be distributed with replication degree m, it is sufficient to generate *m distinct permutations* of these blocks:

$$
\begin{aligned}
Perm_0 &= \{B_0, \ldots, B_i, \ldots, B_{P-1}\} \\
Perm_1 &= \{B_{\mathcal{F}_1(0)}, \ldots, B_{\mathcal{F}_1(i)}, \ldots, B_{\mathcal{F}_1(P-1)}\} \\
\ldots \\
Perm_{m-1} &= \{B_{\mathcal{F}_{m-1}(0)}, \ldots, B_{\mathcal{F}_{m-1}(i)}, \ldots, B_{\mathcal{F}_{m-1}(P-1)}\}
\end{aligned}
$$

where:

- each $\mathcal{F}_k$, $k \in \{1, \ldots, m-1\}$, defines a distinct permutation of $\{0, \ldots, P-1\}$;
- the blocks allocated to processing node p_i are $\{B_i, B_{\mathcal{F}_1(i)}, \ldots, B_{\mathcal{F}_{m-1}(i)}\}$;
- $\forall\ i \in \{0, \ldots, (P-1)\}$ and $\forall\ k, k_1 \in \{1, \ldots, (m-1)\}$, we have that $B_i \neq B_{\mathcal{F}_k(i)}$ and $B_{\mathcal{F}_k(i)} \neq B_{\mathcal{F}_{k_1}(i)}$;
- permutations are built so that the blocks allocated to a generic processing node p_i are *uniformly scattered* on the array 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 node performs iterations which update its primary partition. The second part is dynamic, as it only takes place if a load imbalance occurs. The overhead introduced by the dynamic part of our scheduling strategy is limited, since the static knowledge on the data distribution is effectively exploited. The interactions occurring during the dynamic phase are limited to the processing nodes storing the same blocks, and an effective prefetching mechanism is implemented to overlap communications with computations. Furthermore, when the workloads 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);
     }
 }
```

Fig. 1. Pseudo-code of the scheduling algorithm.

Figure 1 reports the pseudo-code of the scheduling algorithm executed by each processing node p holding a primary partition B and $(m-1)$ secondary partitions. The iterations operating on the primary block of each node are organized into chunks (i.e. the lower and upper limits of a batch of iterations), and inserted into a local queue `Q`. Each node manages another queue `QR` used to store prefetched chunks operating on the $(m-1)$ secondary partitions. At the beginning queue `QR` is empty.

Function `Terminated(prtn_list,Q,QR)` returns `TRUE` *iff* queues `Q, QR` are empty, and p has received a message from all its partners (whose names are stored in the array `prtn_list`), signaling that their local load does not allow them to give work away.
Function `Local_Load(Q,QR)` returns an estimate of the processor's own load, e.g. the number of chunks currently stored in the queues.
Subroutine `Prefetch_Chunks(prtn_list)`, which is executed when the processor load is lower than a given `THRESHOLD`, sends to the active nodes that are the *owners* of the secondary partitions of p, a message requiring chunks to be inserted into `QR`. The active partner to which a chunk request is sent is chosen on a *round-robin* basis. Since most communication overheads are hidden by the prefetching strategy, it is more important not to waste time in making the choice than always making the best choice.
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 nodes holding B as a secondary partition to prevent further requests of chunks. The extracted chunk is soon processed within subroutine `Execute_Chunk(T)`.
Subroutine `Handle_Msg(prtn_list,Q,QR)` receives and manages messages at node p that have arrived from the partners. The messages from partner $\overline{p}$ may

contain:

(i) a chunk that has been previously requested by subroutine `Prefetch_Chunks(prtn_list)`. The chunk is immediately inserted into queue `QR`;
(ii) 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 partner $\overline{p}$ field of structure `prtn_list`.
(iii) the request to move a chunk from queue `Q` of p to queue `QR` of node $\overline{p}$. The request is granted only if node p estimates a load greater than `THRESHOLD`.

In principle, any well-known scheme (such as factoring [7], trapezoid self scheduling [14], guided self scheduling [12], 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 node working on a large chunk may delay the handling of requests from its partners. Such delays increase the overheads and affect the capability of unloaded nodes 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 [15], or high level interrupt handling. Without such mechanisms, uniformly small chunks must be preferred.

4.3 Tuning the implementation template

On distributed memory machines, the effective exploitation of dynamic load balancing strategies generally entails fine tuning several parameters. Our parallel loop implementation template, too, is sensitive to a number of parameters that depend either on the features of the target machine, e.g. the time required to transfer a chunk over the interconnection network, or on the specific problem instance, e.g. the average execution time of loop iterations.
Assuming that the average costs regarding the machine and the problem are known, we need a simple analytical cost model which will allow us to fix the parametric template implementation, i.e. the replication degree, the chunk

[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.

size, and the prefetching threshold that maximize the expected performances. The replication degree is certainly the simplest parameter to fix. As will be shown in the following section, the scheduling overheads are low and any "reasonable" replication degree is well tolerated. Secondary partitions are only exploited when they are useful for balancing the load. When only one subset of secondary partitions is exploited, interactions with the owners of the unexploited partitions are limited to a very simple management of termination.
The smaller the array region is on which time-consuming iterations operate, obviously the higher the probability that blocks all characterized by a low computational load are assigned to a given node thus making it unable to balance the load. Therefore, when the factor of imbalance F is high, the replication degree should be chosen correspondingly large to increase the probability that each node will find a partner from which it can fetch more chunks. However, memory constraints and higher costs if updated values need to be transferred to the corresponding owners have to be taken into account. As an empirical rule, a replication degree lower than the square root of the number of processors used, allows us to balance most non-uniform loops [3]. Higher replication degrees may be useful but only for very particular cases.

Let us now recall some notations and introduce a few new ones for tuning the other parameters:

μ: the average iteration execution time on the target machine;

$\mu_{high} = \frac{t \cdot \mu \cdot D}{d \cdot D} = \mu F$: the average execution time of the $d \cdot D$ more time-consuming iterations (see Section 3);

$\mu_{low} = \frac{(1-t) \cdot \mu \cdot D}{(1-d) \cdot D} = \mu \cdot \frac{(1-t)}{(1-d)}$: the average execution time of the $(1-d) \cdot D$ less expensive iterations;

s: the constant number of iterations included in a chunk;

L: an estimate of the current processor load. L is clearly proportional to the numbers l_L and l_R of chunks stored in queues `Q` and `QR` respectively;

T_{Comm}: the maximum time needed to perform a request for a new chunk on the target parallel machine, i.e. the time needed for processor p to send a message to a node at a distance equal to the diameter of the interconnection network and to receive the corresponding answer.

L_T: the optimal prefetching threshold for the considered problem. Each node should ask for remote chunks and stop giving away local ones when its load becomes lower than L_T.

`THRESHOLD`: the prefetching threshold used by the template implementation. Each node asks for remote chunks and stops giving away local ones when the number of chunks stored in the queues is lower than `THRESHOLD`.

Note that while μ, μ_{low}, μ_{high}, and T_{Comm} depend on the problem instance and

[3] For values of F lower than 10 and up to 128 nodes, we obtained satisfactory performances by adopting replication degrees ranging from 3 to 10.

on the features of the machine, the chunk granularity s and the prefetching threshold `THRESHOLD` can be tuned on the basis of the other parameters to maximize the performances of the parallel loop implementation. In particular, `THRESHOLD` will be derived from the analysis of the possible values of L_T.
Regarding load estimate L, it is clear that the load on each processing node is equal to the costs of all the chunks currently stored in queues `Q` and `QR`. Unfortunately, the exact costs of these chunks are only known after their executions. A heuristic to approximate L must therefore be followed.
Chunk prefetching is clearly exploited only by unloaded nodes, i.e. by those that, on average, execute low-cost iterations on their primary partitions. Remote chunks are in fact never fetched by more loaded processors which, on the other hand, should give away part of their work until they estimate a load greater than L_T. We can suppose that, with high probability, the chunks stored in queues `Q` of unloaded nodes are characterized by a computational cost approximately equal to $s \cdot \mu_{low}$. With the same reasoning, it is likely that on the same nodes, the chunks stored in queue `QR`, have a cost approximately equal to $s \cdot \mu_{high}$. In fact, the chunks operating on secondary partitions have certainly been obtained from more loaded partners.
We can now estimate L for unloaded processing nodes as:

$$L = l_L \cdot s \cdot \mu_{low} + l_R \cdot s \cdot \mu_{high} = s \cdot (l_L \cdot \mu_{low} + l_R \cdot \mu_{high}). \tag{4}$$

Note that when remote chunks have not yet been fetched l_R is equal to zero. On the other hand, loaded processors can estimate L as:

$$L = l_L \cdot s \cdot \mu_{high}. \tag{5}$$

Similar remarks can be made for s and L_T. The computational costs of a chunk transferred across the interconnection network must be greater than the time required to move it. Thus,

$$s > \frac{T_{Comm}}{\mu_{high}}. \tag{6}$$

Moreover, the value for L_T must be large enough to prevent a processor from becoming idle waiting for further remote chunks. The time required to fetch a chunk from a loaded partner may be equal to the communication time T_{Comm} plus the time needed for the partner to execute its current chunk before handling the request. Thus,

$$L_T > T_{Comm} + s \cdot \mu_{high} = T_{Comm} + s \cdot \mu \cdot F. \tag{7}$$

Values for s and L_T cannot, however, be increased without penalties. In fact:

- as noted in the above section, enlarging granularity s results in larger average times required by loaded processing nodes to handle chunk migration requests;
- once all the partners have estimated a load L lower than L_T, chunk transferring is disabled. If, at this time, many chunks are stored in the queues, the probability increases that the time required by each processor to empty the queues differs.

A good trade-off for the choice of these values may therefore be:

$$s = \lceil \frac{T_{Comm}}{\mu} \rceil > \frac{T_{Comm}}{\mu_{high}} \tag{8}$$

and, by substituting Equation (8) in Equation (7),

$$L_T = (F+1) \cdot T_{Comm} \approx T_{Comm} + \lceil \frac{T_{Comm}}{\mu} \rceil \cdot \mu \cdot F. \tag{9}$$

This analysis shows that the implementation of the proposed parallel loop template can be tuned to precisely adapt itself to any particular problem. However, most of these optimizations can only be made if both the factor of imbalance and the load distribution (i.e. t and d) are statically known. This is not the general case. Our implementation template must effectively exploit the parallel machine also when the problem costs are not precisely known. Thus, the template implementation cannot rely upon costs such as μ_{low} and μ_{high}, while the average iteration execution time μ can be assumed known.
The value for s can be still fixed according to Equation (8).
Regarding load estimate L, each processing node knows the values l_L and l_R, and we use their sum as load estimate (this is the value returned by the function $\texttt{Local\_Load}(\texttt{Q}, \texttt{QR}) = l_L + l_R$). However, if both loaded and unloaded nodes adopt the same estimate for L, the value of `THRESHOLD` cannot be the same. In fact, from the previous analysis it results that, on more loaded nodes, L becomes equal to L_T when

$$L_T = I_L \cdot s \cdot \mu_{high}, \tag{10}$$

from which the following value of `THRESHOLD` can be derived:

$$\texttt{THRESHOLD} = l_L = \frac{L_T}{s \cdot \mu_{high}} \approx 2. \tag{11}$$

On the other hand, on less loaded nodes, chunk prefetching should start when

L becomes equal to L_T:

$$L_T = I_L \cdot s \cdot \mu_{low}, \tag{12}$$

from which we can derive the following value of THRESHOLD

$$\texttt{THRESHOLD} = l_L = \frac{L_T}{s \cdot \mu_{low}}. \tag{13}$$

The value determined by Equation (13) still depends from F. Supposing that the average value of F is 5 (e.g. for $t = 0.5$ and $d = 0.1$) we have:

$$\texttt{THRESHOLD} = \frac{L_T}{s \cdot \mu_{low}} = \frac{(F+1) \cdot T_{Comm}}{s \cdot \mu \cdot \frac{(1-t)}{(1-d)}} \approx \frac{F+1}{\frac{(1-t)}{(1-d)}} \approx 10. \tag{14}$$

We therefore implemented the following heuristic. Since a generic node p cannot know whether its load is high or low, at the beginning it assumes that its primary partition is unloaded, and sets THRESHOLD according to Equation (14). At this point two events may occur: either p begins to prefetch chunks because its load becomes lower than THRESHOLD; or p receives a termination message from a partner node. In the first case, p progressively decreases THRESHOLD up to 2 (see Equation (11)) as l_R increases. It thus takes into account that remote chunks stored in QR have higher execution times. In the second case, node p decides that its primary partition is loaded, and immediately decreases THRESHOLD to 2 to continue giving away work.

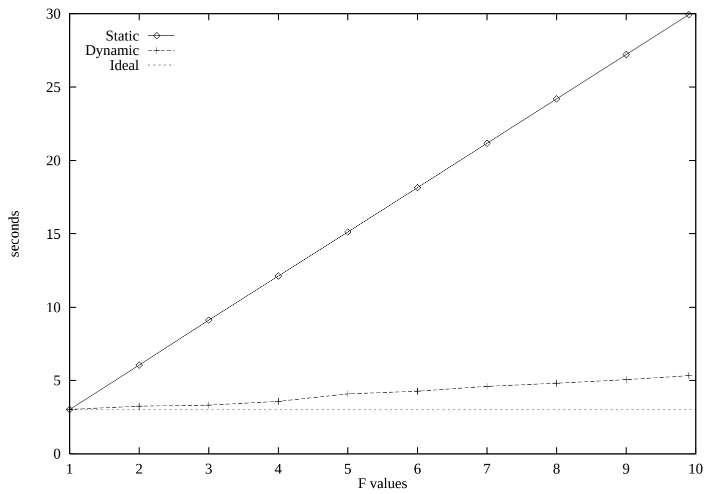

Fig. 2. Completion times on 64 processing nodes as a function of the *factor of imbalance*.

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 $\times$ 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[4]. Iterations were organized into fixed-size chunks of s iterations. The value of s in each test was chosen according to Equation (8), given a value of about one millisecond for T_{Comm} on the nCUBE 2.

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 which was used by setting the replication degree equal to 8 (curve labeled `Dynamic`), and 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. In addition, 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 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. In these tests we exploited a replication degree approximately equal to the square root of the number of processor used. Note that the `Dynamic` curve

[4] We built the input arrays by concentrating the fraction d of expensive iterations on the center of the arrays. However, when the number of nodes used is large with respect to the replication degree the position of the more loaded region becomes irrelevant.

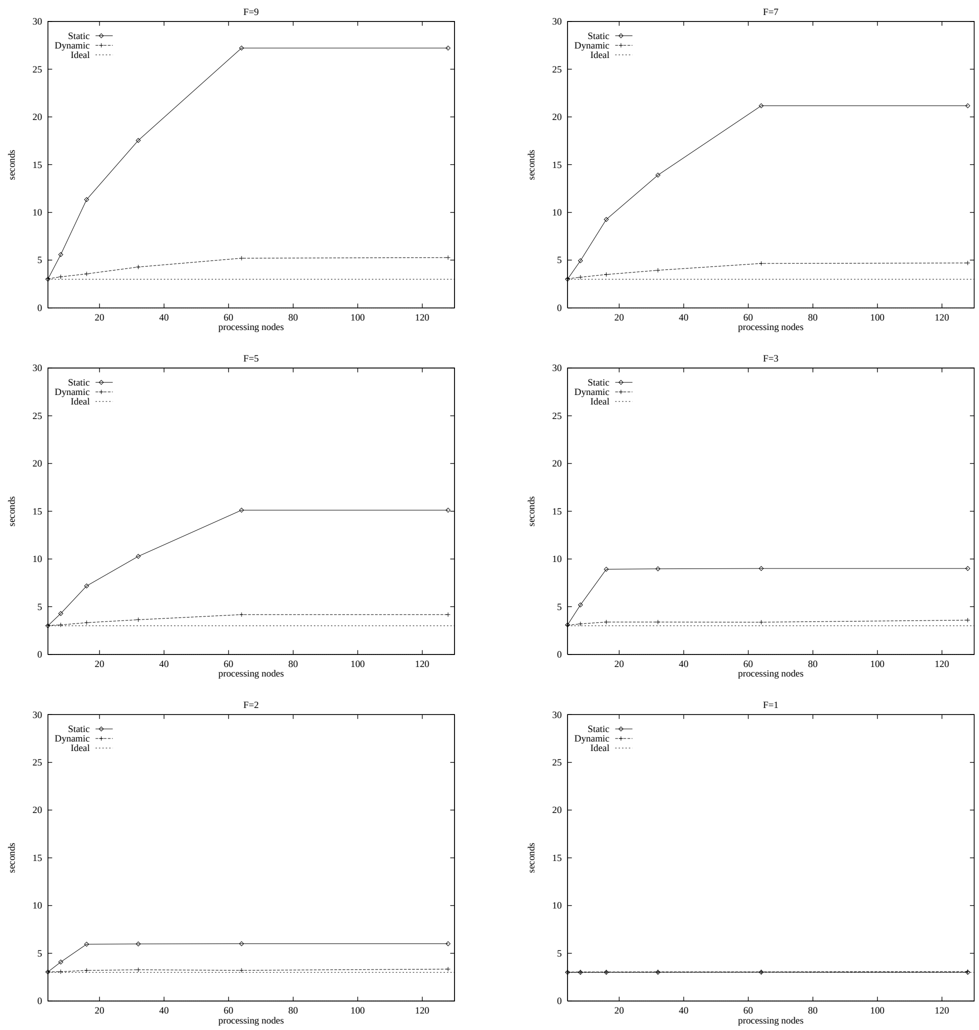

Fig. 3. Comparison between static scheduling and our approach for different *factors of imbalance* as a function of the number of nodes used.

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, on all plots `dynamic` curves have a slope that approaches 0 thus indicating the good scalability of the implementation template.

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 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 nodes 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. Moreover, since the data distribution scheme is established at compile time, this knowledge is used during the dynamic phase to limit the interactions to the processing nodes which store the same partitions. This guarantees the scalability of the technique. 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.
A simple analytical model of load imbalance has been also introduced, which optimizes the performances of the proposed parametric implementation template on the basis of the given problem instance and the related costs on the target machine.
The result of experiments conducted on a 128-node nCUBE 2 multicomputer have been presented and discussed. With our technique we obtained satisfactory performances. 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 equal to 8 on 64 nCUBE 2 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] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A Methodology for the Development and the Support of Massively Parallel Programs. *Future Generation Computer Systems J.*, 8:205–220, 1992.

[3] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, and R.L. While. Parallel Programming Using Skeleton Functions. In *Proc. of PARLE '93 - 5th Int. PARLE Conf.*, pages 146–160, Munich, Germany, June 1993. LNCS 694, Spinger-Verlag.

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

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

[6] 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.

[7] 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.

[8] 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.

[9] V. Kumar, A.Y. Grama, and N. Rao Vempaty. Scalable Load Balancing Techniques for Parallel Computers. *Journal of Parallel and Distributed Computing*, 22:60–79, 1994.

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

[11] 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.

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

[13] D. B. Skillicorn. Models for Practical Parallel Computation. *Int. Journal of Parallel Programming*, 20(2):133–158, April 1991.

[14] 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.

[15] 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.

[16] M.H. Willebeek-LeMair and A.P. Reeves. Strategies for Dynamic Load Balancing on Highly Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, September 1993.

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