A Template for Non–uniform Parallel Loops Based on Dynamic Scheduling and Prefetching Techniques

Salvatore Orlando[†] and Raffaele Perego[‡]

[†] Dip. di Matematica Appl. e Informatica, Università Ca' Foscari di Venezia, via Torino 155, Venezia Mestre, 30173 Italy

[‡] Istituto CNUCE, Consiglio Nazionale delle Ricerche (CNR), via S. Maria 36, Pisa, 56126 Italy

Abstract

In this paper we present an efficient template for the implementation on distributed-memory multiprocessors of *non–uniform parallel loops*, i.e. loops whose independent iterations are characterized by highly varying execution times. The template relies upon a static *blocking* distribution of array data sets to exploit locality, and a hybrid scheduling policy to smooth uneven processor finishing time. It initially adopts a static scheduling technique based on the *owner computes rule*. As soon as a workload imbalance is detected, it exploits a dynamic *receiver initiated* technique to move work towards unloaded processors. Prefetching is used to reduce overheads due to the communications needed to monitor the load, move iterations, and restore the consistency of migrated data. Accurate performance costs of the technique can be derived, thus allowing the template to be used by a compiler to generate well-balanced code for non–uniform parallel loops.

Experiments were conducted on a 64-node Cray T3D, and the performance of the proposed template was compared with the one obtained by using the CRAFT-Fortran language (an HPF-like language).

1 Introduction

Data parallelism has been recognized as the most common form of parallelism in many scientific compute-intensive applications. The effective exploitation of data parallelism is thus the key to achieving considerable speedups of these applications. In imperative languages, *parallel loops* operating on arrays are the usual constructs to express data parallel computations. Therefore a lot of research on parallel languages is focused on finding good run-time supports for the execution of loops on shared-memory symmetric multiprocessors and, recently, on distributed-memory multiprocessors.

One of the main issues in implementing parallel loops is the allocation of data with respect to the schedule of iteration. In UMA shared-memory multiprocessors, where, in principle, shared data are at the same distance from any processor, the usual implementation schemes rely on completely dynamic self-scheduling techniques, based on the existence of a global queue of iterations [13, 15, 11, 6]. The introduction of caches makes this scheme unsuitable even for UMA multiprocessors, since it does not guarantee the exploitation of locality [11]. In distributed-memory multiprocessors, data distribution and iteration scheduling are also more strictly related. Data parallel languages such as HPF [4] rely on user-provided directives to derive data distribution. Scheduling is usually static, and depends on the data distribution. The *owner computes rule* [17] is normally used to determine static schedules from data distribution. Programmers can specify alignments of arrays to each other, and distribution of arrays to processors according to some standard policies, i.e. cyclic and blocking. The main purpose of the blocking distribution directive is to give the compiler more information which can be used to produce a code which exploits locality, i.e. a code which is characterized by limited communication overheads. Conversely, cyclic distribution is good for problems where locality is less important than load balance issues. This occurs when the work required for each section of the data array varies significantly. Standard HPF also allows data mapping to be changed at run-time, by using redistribution and realign "executable" directives.

The choice of the best data distribution is currently up to programmers. It is known that, depending on some features of parallel loops (mainly, array references), the problem of finding an optimal data layout may be in NP [8]. On the other hand, it is likely that, for the cases where data references are either constant or are simple linear combinations of indexes, future compilers will be sophisticated enough to automatically align and distribute arrays, and thus obtain acceptable performances. These smart compilers will not be able, however, to solve cases in which, by using simple blocking or cyclic data distribution schemes, only one of the two contrasting goals is achieved, i.e. either locality or load balancing. These goals may in fact conflict with each other because one may need a blocking data distribution strategy to exploit locality, but also a cyclic data distribution to balance the workload among the processors.

Moreover, no automatic and semi-automatic *static tools* will be able to devise optimal data distributions when array references and computational loads of each iteration change at run-time in an unpredictable way. Note that, although our HPF compiler supports *dynamic data mapping directives*, the unpredictable behaviour of programs prevent us from using them effectively.

This paper presents a template to efficiently implement *non-uniform parallel loops* on distributed-memory multiprocessors. The technique is explicitly designed to smooth at run–time any *load imbalance* deriving from processor workloads characterized by highly varying execution times. Note that we do not address the general load balancing problem, but we study this problem in the context of a specific form of parallelism, namely parallel loops. As shown in [14], the restriction of the computational model to specific forms of parallelism allows us to devise much more effective implementations by adopting a parallel programming methodology known as template-based [2] or skeleton-based [3].

Our template relies upon a blocking data distribution to exploit locality, and combines static and dynamic policies to obtain well-balanced iteration schedules. We adopt, as far as possible, a static scheduling policy, and introduce dynamic scheduling and data movements among processors only if a load imbalance occurs. Two different version of the template are presented. The first version is optimized for *iterated parallel loops*, i.e. parallel loops nested in a sequential one, for which it is necessary to maintain the coherence of the data among successive iterations of the outer sequential loop. Both data and iteration indexes are in this case moved across the interconnection network. The second version is specifically designed for parallel loops which are not nested or which do not modify the accessed data. In this case static partial replication of the data set is exploited to minimize the run–time overhead of the technique [12].
An effective prefetching mechanism considerably reduces overheads by overlapping communications with computation. Communications are needed to monitor the load, to transfer work, and to restore consistency of migrated data. Moreover, a fine control over data coherence, implemented by using a full/empty bit like technique [1], permits the removal of all the synchronizations between successive iterations of the parallel loop.

The paper details the implementation template, and presents the experiments that were conducted to validate the idea. The experiments were run on a 64 node Cray T3D, by employing synthetic benchmarks that model several typical cases of load imbalance. We chose a benchmark which is a real challenge for our technique, as it favours static scheduling based on a cyclic data distribution scheme. In fact, we consider a data parallel loop in which each element in the array is obtained by applying a function whose only input parameter is the old value of the same array element. Thus, independently of the data distribution, processors do not need to communicate to exchange data. We show that, even in this limit case, our technique is able to obtain execution times shorter than the HPF-style implementation. To compare our technique with an HPF-style implementation, we used the Cray CRAFT-Fortran, whose language directives and compilers have been adapted in part from Rice University's Fortran-D project [5] and Vienna Fortran [17]. We can thus assume that the CRAFT-Fortran is very similar to the HPF standard, as specified by the HPF Forum.

The paper is organized as follows. Section 2 briefly surveys related work. Section 3 describes the proposed implementation template. Section 4 presents a simple model of loop load imbalance used to derive the benchmarks. Assessments of the experiment results are reported and discussed in Section 5. In Section 6 we draw some conclusions.

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 [13]. *Trapezoid Self Scheduling* [15] was proposed by Tzen and Ni to reduce the number of synchronizations by linearly decreasing the chunk size. Hummel, Schonberg and Flynn presented *Factoring* [6], 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. 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:

1. **Processor Load Evaluation**, how each processing node estimates its own load if needed;
2. **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;
3. **Task Migration Strategy**, how the source and the destination of a task migration are determined.

They also presented a technique called *Receiver 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 requests to some of its partners chosen on a simple inexpensive round-robin basis.

3 The technique

In distributed-memory environments a given parallel loop iteration can be scheduled on a given processing node only if that node holds a copy of the data needed to execute the iteration. The solution of replicating the full loop dataset should, in principle, make any dynamic scheduling technique more efficient, since it reduces the amount of information exchanged between processors. However, the complete data replication may be unfeasible due to prohibitive memory requirements, and to the costs of keeping all data copies coherent. Of course, there is no coherence problem if the data set is read-only. Alternative solutions to reduce communication overheads of dynamic scheduling policies are based on partial replications. When the coherence constraint is loose or can easily be managed, dynamic scheduling techniques that rely upon the partial replication of the data can be successfully exploited [12, 10]. On the other hand, if the coherence constraint is tight, for example we have a parallel loop which is nested inside a sequential loop and which updates the replicated data set, partial replication has no benefits since the coherence for the distributed data copies has to be managed.

Our technique relies upon a blocking partitioning of the array data set to exploit locality. Accessed arrays, which may be *aligned* to an abstract topology, are partitioned into P contiguous blocks, and each block is statically assigned to one of the P processors. Iterations operating on each partition are subdivided into *chunks* of g iterations, and assigned to the various processors according to the owner computes rule. Each processor has a local queue Q to store its own chunks.

This first assignment determines the static part of our scheduling technique. In fact, at the beginning, and until a load imbalance has been detected, chunks are statically scheduled by fetching them from Q.

The dynamic part of the scheduling policy starts when a processor understands that queue Q is *becoming empty* and requires chunks indexes and data to be transmitted over the interconnection network. With respect to a generic processor p_i, we call the chunks initially stored in the queue Q of p_i *local*, while we call the chunks moved at run–time to p_i *remote*.

To limit overheads, each node may ask a limited number of other processors for further chunks. We determine this set of *partner* processors a priori, and call its cardinality *partnership degree*.

Partnership choice. The partnership degree determines the number of partner processors that can be asked for further work. These sets of partners are statically fixed for each processor. If m is the partnership degree, each processor p_i has m partners, with the additional property that p_i is assigned as a partner to a set of exactly m other processors.

More formally, to generate the various sets of partners of n processors $p_1, \ldots, p_i, \ldots, p_n$, it is sufficient to generate m *distinct permutations* of these processors:

$$\begin{array}{lllll} p_1, & \cdots & p_i, & \cdots & p_n \\ p_{f_1(1)}, & \cdots & p_{f_1(i)}, & \cdots & p_{f_1(n)} \\ & \cdots & & & \\ p_{f_m(1)}, & \cdots & p_{f_m(i)}, & \cdots & p_{f_m(n)} \end{array}$$

where

- each f_k, $k \in \{1, \ldots, m\}$, defines a distinct permutation of $\{1, \ldots, n\}$;
- the set of partners of a generic processing node p_i is $\{p_{f_1(i)}, \ldots, p_{f_m(i)}\}$;
- $\forall\ i \in \{1, \ldots, n\}$ and $\forall\ k, k_1 \in \{1, \ldots, m\}$, we have that $p_i \neq p_{f_k(i)}$ and $p_{f_k(i)} \neq p_{f_{k_1}(i)}$.

A desiderable property of the partner set choice is that an unloaded processor should be able to find, with high probability, loaded processors within its own partner set. This depends, of course, on the features of the input data set. Since we adopt a blocking data distribution, the cases which may cause higher load imbalances occur when most computational load is concentrated on some (unknown) regions of the data set. The worst case might occur, for example, when the most part of the load is concentrated on a single small block of the data set. Thus, it is important to choose the m partners of each processor so that the data partitions assigned to these m partners are evenly distributed on the whole array data structure.

Load balancing strategy. Chunk migration decisions are taken on the basis of local information only. Two partner processors decide the profitability of a chunk migration by considering: **(1)** the number of local chunks not yet scheduled, **(2)** the average cost of each local chunk, and **(3)** the cost of chunk migration which depends on the specific parallel machine.
Since it is the *receiver* of a remote chunk that starts chunk migration, our technique can be defined as a *receiver initiated* technique [9, 16]. According to the framework proposed by Willebeek-LeMair and Reeves [16], the technique can be characterized as follows:

- **Processor Load Evaluation.** Since the first part of our technique is static, the average cost of local chunks can be determined by a very simple code instrumentation, which accumulates the time spent by chunk executions. The architecture we have used, a Cray T3D, is based on the Digital Alpha processor, which provides a simple machine instruction to read the clock cycle count, thus reducing the overhead of code instrumentation.

We compute the *local load* of a processor as the number of chunks in Q that have not been scheduled yet, times the average execution cost of the chunks already executed. A *Threshold* value for the *local load* is used to decide when processor load is becoming low, and depends on the specific machine, i.e. on the specific cost of chunk migration. Roughly speaking, this *Threshold* is low for machines where communications are very efficient, while it must be higher otherwise. Note that the value of *Threshold* must be high enough to assure that chunks are prefetched so as to minimize idle waiting times of unloaded processors.

- **Load Balancing Profitability Determination.** Each processor tries to balance the load, thus starting the dynamic part of the scheduling technique, on the basis of local information only. It compares the value of its *local load* with the machine-dependent *Threshold*. When the computed local load becomes lower than *Threshold*, the processor begins to ask its partners for remote chunks. The same comparison of *local load* with *Threshold* is used by the partner to decide chunk migration. A processor p_j, which receives the request of chunk migration from a processor p_i (thus, p_j is a partner of p_i), will actually grant the request only if its *local load* is higher than *Threshold*.

  Note that unloaded processors begin to prefetch chunks from the partners when they have still many local chunks in Q, while loaded processors stop granting chunk migration requests when their queues are nearly empty. This happens due to the *Processor Load Evaluation* strategy. On average, in fact, local chunks executed on loaded processors take larger to execute than those of unloaded processors.

  To reduce overheads deriving from requests for remote chunks which cannot be served because the *local load* of the partner has become low, each processor, when its load becomes lower than *Threshold*, sends a *termination message* to all the other processors which could ask it to migrate chunks. The arrival of all the termination messages also determines the end of execution of the parallel loop. When the local queue Q of a processor p_i is empty, p_i does not have migrated chunks to execute, and p_i received a termination message from all of its partners, then it can locally terminate its participation in the parallel loop execution. Note that, when p_i terminates, since its queue Q is empty, it has certainly already communicated its termination to all the processors of which it is partner.

- **Task Migration Strategy.** The source (*sender*) of a chunk migration is determined by the destination (*receiver*) of the chunk. The *receiver* selects the processor which it asks for further chunks by using a *round-robin* policy within the set of its *partners* that have not yet communicated their *termination*. This criterion was chosen because of its simplicity. As discussed in the paper by Kumar et al. [9], it is in fact more important not to spend time in making the decision than to always make the best decision.

Data replication. Our technique may exploit data replications to avoid paying, when a chunk is migrated, the cost of data transfer. Data replication may be useful, for example, when a data set is read only, or, as we will see below, when the parallel loop is not nested. On the other hand, when the coherence constraint is tight, for example, we have a parallel loop which is nested inside a sequential loop and which updates the data set, data replication has no benefits because of the high cost due to restoring the coherence of the distributed data copies at the end of each execution of the parallel loop. The choice of transferring both chunk indexes and related data is, in this case, more efficient.

Extending the technique to manage data replication is trivial: data partitions are statically replicated according to the partnership choice. This means that each processor, besides its *primary data partition*, stores a copy of the various partitions primarily assigned to its partners. We call them *secondary data partitions* assigned to a given processor. When a processor p_i receives a remote chunk from a partner, it already holds *secondary data partitions* containing the data on which chunk iterations must be executed.

SPMD Code of the Template. Figure 1 shows the SPMD implementation of our template, in particular, the pseudocode executed by a generic processor p_i, $i \in \{1, \ldots, n\}$. As can be seen, two distinct arrays `partners[]` and `is_partner_of[]` characterize the code of each processor p_i. The former array contains the set of the *partner* processors of p_i, $\{p_{i_1}, \ldots, p_{i_m}\}$, $m < n$, where m is the *partnership degree* of our technique. These are the processors to which p_i will, if necessary, ask for further work if a load imbalance occurs.

Since p_i is assigned as a partner to a set of exactly m other processors, this last set of m processors is stored in the latter array `is_partner_of[]`. This array is used by p_i to find out from which processor it might be asked to migrate work.

Two queues are allocated to each processor, the local queue Q, which includes the statically allocated iteration chunks, and the remote queue QR, which will be used to store remote chunks received from more loaded partners. Finally, the parameter K determines the number of times the parallel loop must be iterated.

The SPMD code shown in Figure 1 is a *while* loop, iterated for K times, whose body corresponds to the execution of an *instance* of the parallel loop.

The function *Terminated(Q, QR, partners)* checks the termination condition: p_i terminates an instance of the parallel loop as soon as its queues Q and QR are empty, and it has received a termination message from all the members of `partner[]`. The function *My_load(Q)* estimates the current local *workload* of a processor p_i. It simply returns the product of the chunks still stored in Q times the average chunk execution cost. This cost is determined by monitoring the execution of the local chunks which are statically scheduled during the first phase of our technique. At the beginning, when no chunk executions have been monitored, this cost is initialized with a high value to avoid wrong chunk requests and migrations.

The prefetching policy adopted to reduce overheads of chunk migration is driven by a machine-dependent *Threshold* parameter: a processor asks a partner for a remote chunk (*Prefetch_Chunk (partners)*) only if its *local load* is lower than *Threshold*. Likewise, the partner process serves the requests only if its own *local load* is greater than *Threshold*. We limit the number of chunk requests sent by a processor and not yet granted. Thus, *Prefetch_Chunk(partners)* sends a request message only if this limit has not been reached. *Prefetch_Chunk(partners)* employs a simple local *round-robin* policy to select the next partner to be asked for a remote chunk. Note that the next partner is selected from those which still have not communicated that their load has be-

```
Iterated_Parallel_Loop (queue Q, process partners[ ],
                        is_partner_of[ ], int K)
QR: queue;
T: chunk;
begin
  for i = 1 to K do
    Initialize_Data_Structures ();
    while (not Terminated (Q, QR, partners)) do
      if (My_Load (Q) < Threshold) then
        Prefetch_Chunk (partners);
        if (* My_Load (Q) < Threshold for the first time *) then
          Comm_unloaded (is_partners_of);
        endif
      endif
      T = Extract(Q);
      if (Empty (T)) then
        T = Extract (QR);
        Execute (T);
        Send_Results (T);
      else
        Execute (T);
      endif
      Request_Handler ();
    endwhile
  endfor
end
```

Figure 1: Pseudo-code of the scheduling algorithm.

come smaller than *Threshold*. Each processor communicates this event to all the processors from which it may receive a chunk request by means of the subroutine *Comm_unloaded (is_partner_of)*. Note that these communications are also used to implement a distributed *termination protocol*, so that we call them *termination messages*.
The function *Extract()* returns a chunk from a queue. The subroutine *Execute(T)* executes a chunk T, and, if T is local, measures the time taken to complete it. This measure is used to evaluate the average cost of local chunk executions.

We adopt an asynchronous *coherence protocol* to migrate updated data from the processor that actually performed the chunk to the processor that is the owner of the corresponding data partition. The updated data are transmitted by the subroutine *Send_Results (T)*. A *full/empty-like* technique is used by our coherence protocol to avoid processing invalid data. When processor p_i sends a chunk b_j to p_{i_k}, it sets a flag indicating that the data to be modified by b_j are no longer valid. The next time p_i needs to access the same data, for example during the next iteration of the outer sequential loop, p_i checks the flag and, if the flag is still set, waits for the updated data from node p_{i_k}. Such control of data coherence does not require barrier synchronizations at the end of each execution of the parallel loop. Note that, by using this asynchronous coherence protocol, it is very unlikely that a processor is blocked while waiting for an unset flag associated with a local chunk. In fact, most coherence protocol communications related to a given parallel loop instance are overlapped with chunk computations belonging to the same loop instance. Moreover, when a new instance of the parallel loop is initiated, residual coherence communications, i.e. those related to the previous loop instance, are overlapped with the computations of the first chunks of the new instance of the loop. Since our technique is based on a first static scheduling phase, and the same chunk scheduling order is followed at each parallel loop execution, it is likely that the first scheduled chunks do not have to wait for any full/empty flag to be unset.

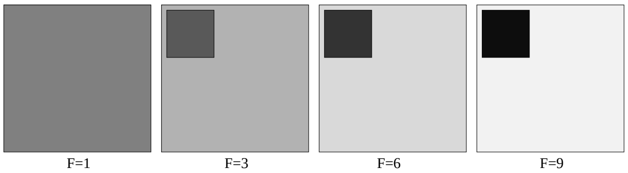

Figure 2: Different arrays characterized by different factors of imbalance (F), with $d = 0.1$. The grey levels used to fill the squares represent the workload associated with array elements, where a darker grey stands for a heavier workload.

Finally, the subroutine *Request_Handler(Q, QR)*, whose code, for the sake of simplicity, is not shown in detail, performs many of the above tasks. This subroutine handles all the messages that can arrive at a processor, i.e. requests for more work from an unloaded processor (this may require a chunk to be extracted from Q and sent to the unloaded processor), coherence messages transporting updated data, further chunks sent by partner processors (these chunks are inserted in QR), and service messages which signal that the local queue of a partner process is emptying (termination messages). The behavior of subroutine *Request_Handler(Q, QR)* clearly differs depending on whether data replication is exploited or not. If exploited, only chunk indexes have to be transmitted to grant a chunk migration request. Conversely, when the template entirely relies upon run–time data transfer, *Request_Handler(Q, QR)* also manages the sending/receiving of the data needed to execute remote chunks.

4 A model of load imbalance and the derived synthetic benchmark

The benchmark that we have used to validate the idea is a parallel loop which operates on a large bi-dimensional array of floating point numbers. The computation performed by the parallel loop is simply a function `COMP` that is applied to all the elements in the array to produce the new ones. The execution time of each function instance is directly proportional to the value of the processed array element. If μ is the average execution cost of parallel loop iterations, and D is the number of iterations, $T = D \cdot \mu$ is the total workload of the parallel loop. If the parallel loop is iterated K times, the total workload is $K \cdot T$. The HPF-like code of our benchmark follows:

```
DO h= 1,K
   FORALL (i= 1:N,  j= 1:N)
      A[i,j] = COMP(A[i,j])
   END FORALL
END DO
```

Note that, to keep constant the computational load associated with `COMP()`, the function returns the same value it received as its input parameter. The values stored in `A[]` determine the workload, which may be unbalanced.

The model we used to generate the various arrays `A[]`, each corresponding to a problem characterized by a given unbalanced workload, is based on two parameters, d and t. The parameter d, $0 < d < 1$, determines the fraction of the whole array on which the per-element execution time is greater than μ, and t, $0 < t < 1$, corresponds to the fraction of T concentrated of the portion of the array identified by d. Therefore, it follows that $t > d$.

Note that the imbalance is directly proportional to t for a given value of d, since larger values of t correspond to larger fractions of T concentrated of the portion of the array identified by d. Similarly, the imbalance is inversely proportional to d for a given value of t. From these remarks, we can derive F, called *factor of imbalance*, defined as

$$F = \frac{t}{d} \tag{1}$$

F is directly proportional to the unbalanced workload that characterizes the array.

Figure 2 represents four two-dimensional arrays characterized by different values of F (built by keeping $d = 0.1$ fixed), which we used for our experiments.

5 Evaluation of the results

We tested our template on a 64 node Cray-T3D by using the MPI message passing library, release CRI/EPCC 1.3, developed at the Edinburgh Parallel Computing Centre in collaboration with Cray Research Inc.

As described in Section 4, we chose a synthetic benchmark which is a real challenge for our technique. In fact, by considering a parallel loop in which each element in the array is obtained by applying a function whose only input parameter is the old value of the same array element, we do not take advantage of data locality deriving from adopting a *blocking* data distribution. The processors do not need to communicate to exchange non–local data even if a cyclic data distribution scheme is adopted. This choice should favour a HPF–like language implementation, which adopts a static scheduling approach based on a cyclic data distribution.

In order to stress the scalability of our template, all the experiments whose results are presented in this paper exploited all the 64 nodes of the T3D computer used. The size of the problem (1024×1024 floating point numbers), and, correspondingly, the size of the array blocks (128×128) were fixed. Another fixed parameter was the *partnership degree* (40). We experimentally evaluated that the adoption of this high value for the partnership degree is needed for higher values of F, but also that overheads introduced by this high partnership degree for smaller F's are negligible.

The machine-dependent *Threshold* parameter was fixed to 6 $msec$, while each unloaded processor can have, at each time, a maximum number of three ongoing prefetching messages.

The implementation parameters that we varied to conduct the experiments on different benchmarks, each characterized by a distinct F, are the *average iteration execution time*, μ, and the *chunk size* g.

Since a processor sends/receives messages between the execution of two successive chunks, μ and g influences the rate at which a node injects/draws messages into/from the network. Moreover, we have to consider that the network traffic generated by our template may easily produce *hot-spots*, so that a smaller injection rate into the network would be desiderable. In fact, since we are assuming that most of the total workload is concentrated on a few processors, these processors will receive a lot of requests asking for chunk migrations, and a lot of coherence messages returning modified data.
Thus, by adopting large values of g we should, in principle, reduce the network traffic and obtain a better performance. On the other hand, in case data have to be migrated along with chunks, or coherence messages must be returned to the owner of a chunk, g also influence the size of messages. Moreover, very large values of g reduce the ability of a dynamic scheduling technique to balance the workload. Thus, there should be a tradeoff value for g. To make our template usable by a compiler, we have to find a tradeoff for g with respect to typical values of μ and F. It is difficult to derive the right values of g analytically, but we can use the benchmark program to derive them. However, as we will see below, for large values of μ changes in the value of g slightly influences the final performance.

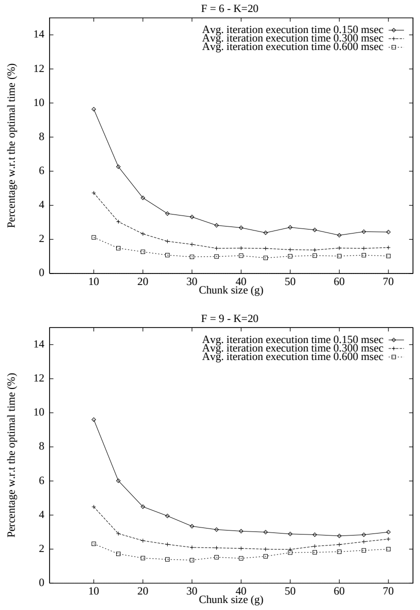

Figure 3: Curves plotting, as a percentage with respect to the theoretical optimal execution time, the differences between the performances of the template exploiting data transfer. The curves plot these percentages as a function of g for $F = 6$ and $F = 9$, $\mu = 0.15$ $msec$, $\mu = 0.3$ $msec$, $\mu = 0.6$ $msec$.

5.1 Nested parallel loops

When the coherence constraint is tight, for example we have a parallel loop which is nested inside a sequential loop and which updates the data set, partial data replication has no benefits because of the high cost of managing the coherence for the distributed copies of the blocks. Thus, the template that transfers at run–time the data needed is in this case more efficient. Moreover, the fine control over data coherence, implemented by using a full/empty bit like technique [1], permits the removal of all the synchronizations between successive iterations of the parallel loop. The absence of barrier synchronizations allows most of the coherence communications to be overlapped with the computations of the next iteration of the outer sequential loop. This explains the very good performances obtained by our implementation on nested parallel loops. Figure 3 shows some results obtained by executing 20 iterations ($K = 20$) of the same parallel loop. Each curve corresponds to a distinct problem instance. In particular, the first plot shows three curves corresponding to three problems characterized by the same $F = 6$ ($d = 0.1$ and $t = 0.6$), and by $\mu = 0.15$ $msec$, $\mu = 0.3$ $msec$ and $\mu = 0.6$ $msec$, respectively. The second plot shows similar curves for $F = 9$ ($d = 0.1$ and $t = 0.9$).

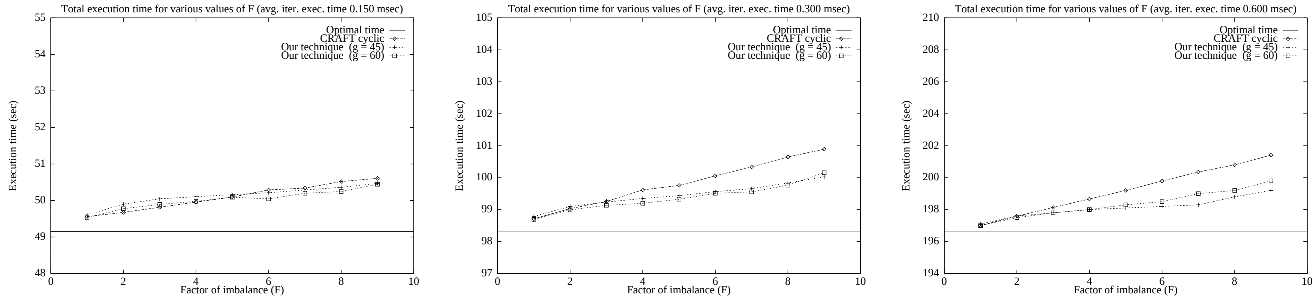

Figure 4: Execution times for different values of μ as a function of F, and comparison with the optimal execution time and a CRAFT-Fortran program version.

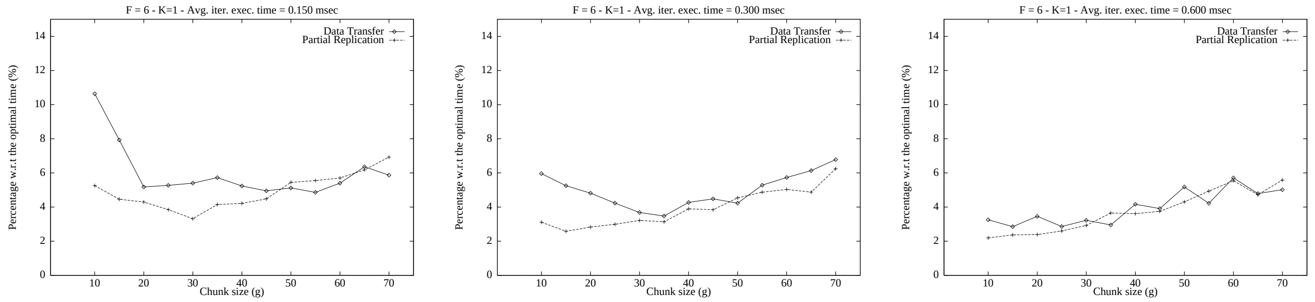

Figure 5: Curves plotting, as a percentage with respect to the theoretical optimal execution time, the difference between the performances of the template exploiting or not exploiting partial data replication. The curves plot these percentages as a function of g for $F = 6$, $\mu = 0.15\ msec$, $\mu = 0.3\ msec$, $\mu = 0.6\ msec$.

The curves plot, as a percentage with respect to the theoretical optimal execution time, the difference between the actual execution time and the optimal time. The minimum of these curves corresponds thus to the best execution time.

Note the tradeoff value for g: for $F = 9$ and $\mu = 0.15\ msec$, $\mu = 0.3\ msec$ and $\mu = 0.6\ msec$, the best values of g are 60, 50 and 30, respectively. If we adopt either smaller or larger g, the execution times slightly increase. This increase is, however, negligible thus allowing the template to work efficiently for many different values of g. This property is very desiderable as it makes the performances of the template less sensitive to the choice of g. Note that the best value for g depends both on the factor of imbalance F and the average iteration execution time μ which are, in general, unknown at compile time.

In addition, note that the larger the average iteration execution time μ, the closer the corresponding curves are to the optimal time. This happens because, for larger values of μ, smaller injection rates are exploited by the template, and the ratio between computation and communication increases.

Figure 4 shows the actual execution times obtained for problems characterized by $K = 20$, $F = \{1, \ldots, 9\}$ and $\mu = 0.15\ msec$, $\mu = 0.3\ msec$, $\mu = 0.6\ msec$. Each curve was obtained by adopting a value for g equal to 45 or 60 iterations. These values were chosen on the basis of the curves in Figure 3. Figure 4 also compares the various execution times obtained by our template with the theoretical optimal execution time, and with the execution time obtained by the CRAFT-Fortran version of the same loop when a cyclic array distribution is adopted. We have not shown the large execution times obtained by the CRAFT-Fortran version when a blocking distribution is adopted. They are about F times the theoretical optimal execution times. For example, for $F = 9$ and $k = 20$, the execution times of the CRAFT-Fortran program exploiting a blocking distribution are 446, 895, and 1776 seconds, for problems characterized by $\mu = 0.15\ msec$, $\mu = 0.3\ msec$, $\mu = 0.6\ msec$, respectively. The execution times obtained by our template are thus equivalent or better than the CRAFT-Fortran ones. Moreover, our template does not require the programmer to choose the best data distribution, since we adopt a blocking data mapping which, depending on the program features, may be very useful as it allows locality to be effectively exploited.

5.2 Single parallel loops

This section presents some results obtained by running single parallel loops ($K = 1$) on different benchmarks. In this case we used two templates. The first adopts data replications and does not maintain the coherence of the primary data partitions. With respect to this template, the chunk size g only influences the injection rate of messages into the network, but not the message size, which is small and constant. The second template transfers data along with chunks. The results produced by the remote chunk execution are, in this case, returned to the sender of the chunk.

Figure 5 shows some curves that plot, as a percentage with respect to the theoretical optimal execution time, the difference between the performances of the template exploiting or not exploiting partial data replication. The curves plot these percentages as a function of g for $F = 6$, $\mu = 0.15\ msec$, $\mu = 0.3\ msec$, $\mu = 0.6\ msec$. As can be seen, the reduction of communication overheads resulting from the exploitation of partial data replication is sensitive. Moreover, by exploiting partial data replication the best percentages are obtained for smaller values of g, e.g. 15 instead of 35 for $\mu = 0.3\ msec$. This happens because of the smaller

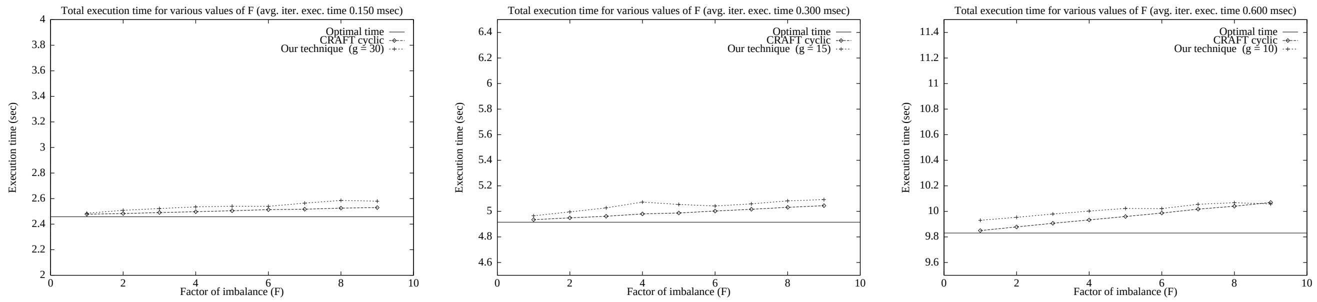

Figure 6: Execution times for different values of F of the template exploiting partial data replication, and comparison with the optimal execution time and the execution time of the CRAFT-Fortran program adopting a cyclic array distribution.

messages travelling on the interconnection network which reduce the cost/benefits ratio of chunk migration.

Figure 6 shows the actual execution times obtained with the template exploiting partial data replication for problems characterized by $F = \{1, \ldots, 9\}$ and $\mu = 0.15$, $\mu = 0.3$ and $\mu = 0.6$. The plots compare the template execution times with the theoretical optimal execution time, and with the execution time obtained by the CRAFT-Fortran version of the same loop exploiting a cyclic array distribution. Note that the execution times achieved by our implementation are comparable with the CRAFT program ones, regardless of the different data distribution scheme adopted.

6 Conclusions

A template for the implementation on a distributed–memory multiprocessor of non–uniform parallel loops has been presented and discussed. It relies upon a static *blocking* distribution of the data sets and a hybrid scheduling policy which initially adopts a static technique to distribute the loop iterations among the processing nodes. As soon as a workload imbalance is detected, the template exploits a dynamic *receiver initiated* technique to move iterations and data towards unloaded processors. An efficient prefetching mechanism is used to reduce overheads due to the communications needed to monitor the load, move iterations, and restore the consistency of migrated data.

As a consequence of an accurate implementation, the template performances are better than or comparable with those obtained with a semantically equivalent CRAFT-Fortran program. A *cyclic* distribution, which causes the loss of data locality, must be adopted by the CRAFT-Fortran program to achieve acceptable performances. The possibility of exploiting data locality deriving by the use of a *blocking* distribution is instead one of the main goals of our technique, even if the benchmark we used to validate the template does not take advantage of it. Note that, if our template is adopted, programmers do not have to be responsible for choosing the best data distribution scheme for each problem instance.

Accurate performance costs of the technique can be derived, thus allowing the template to be used by a compiler to generate well-balanced schedules for non-uniform parallel loops.

As a further optimization, we plan to substitute the MPI library used in the first implementation with Cray T3D low-level cooperation mechanisms [7] in order to further reduce template overheads.

References

[1] R. Alverson et al. The Tera computer system. In *Proc. of the 1990 ACM Int. Conf. on Supercomputing*, pages 1–6, 1990.

[2] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: a Structured High-level Parallel Language and its Structured Support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995.

[3] J. Darlington et al. Parallel Programming Using Skeleton Functions. In *Proc. of 5th Int. PARLE Conf*, pages 146–160, Munich, Germany, June 1993. LNCS 694, Spinger-Verlag.

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

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

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

[7] V. Karamcheti and A. A. Chien. A Comparison of Architectural Support for Message Passing on the TMC CM–5 and Cray T3D. In *Proc. of 22th ACM ISCA*, pages 298–307, June 1995.

[8] C. W. Kessler. Pattern-driven automatic program transformation and parallelization. In *Proc. 3rd EUROMICRO Work. on Parallel and Distr. Proc.*, pages 76–83, Jan. 1995.

[9] V. Kumar, A.Y. Grama, and N. Rao Vempaty. Scalable Load Balancing Techniques for Parallel Computers. *J. of Parallel and Distr. Comp.*, 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 Trans. on Parallel and Distr. Systems*, 5(4):379–400, April 1994.

[12] S. Orlando and R. Perego. Exploiting Partial Replication in Unbalanced Parallel Loops Scheduling on Multicomputers. *Microprocessing and Microprogramming*, March 1996. to appear.

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

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

[15] 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.

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

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