UNIVERSITÀ CA' FOSCARI DI VENEZIA
Dip. di Matematica Applicata ed Informatica
Technical Report Series on Computer Science

Technical Report CS-97-9

April 18, 1997

Salvatore Orlando and Raffaele Perego

A Comparison of Implementation Strategies for Non–Uniform Data–Parallel Computations

Corso di Laurea in Informatica
Via Torino, 155 – 30173 – Venezia Mestre

A Comparison of Implementation Strategies for Non–Uniform Data–Parallel Computations*

Salvatore Orlando[†], Raffaele Perego[‡]

April 18, 1997

Abstract

Data–parallel languages allow programmers to easily express parallel computations by means of parallel loop constructs or collective operations. To reduce overheads, the compiler partitions the computations among the processors at compile–time, on the basis of the static data distribution suggested by the programmer. When execution costs are non–uniform and unpredictable, some processors may be assigned more work than others. Workload imbalance can be mitigated by cyclically distributing data and associated computations, or by employing dynamic strategies which build a more balanced schedule at run–time, on the basis of the actual execution costs.

This paper discusses static and hybrid (static + dynamic) strategies which can be used to balance the workloads derived from the execution of non–uniform parallel loops. A multi-dimensional flame simulation kernel has been used to evaluate different implementation strategies on a Cray T3E. We fed the benchmark code with synthetic input data sets built on the basis of a load imbalance model and we report and compare the results obtained.

*This paper has been submitted for publication to the Journal of Parallel and Distributed Computing.

[†]Department of "Matematica Applicata e Informatica" of the "Università Ca' Foscari di Venezia", via Torino 155, Venezia Mestre, 30173 Italy. E-mail: orlando@unive.it

[‡]"Istituto CNUCE" of the "Consiglio Nazionale delle Ricerche (CNR)", via S. Maria 36, Pisa, 56126 Italy. E-mail: r.perego@cnuce.cnr.it

1 Introduction

High–level data parallel languages make it easier to write data parallel programs for both shared and distributed–memory multiprocessors. Exploiting the results of previous research projects such as the Fortran D project at Rice and Syracuse universities [6], and the Vienna Fortran project at the University of Vienna [26], a consortium of academic and industrial partners (participating in the *High Performance Fortran Forum*) defined HPF [3]. HPF is a set of data parallel extensions to Fortran 90 that provide support for programming parallel machines where the distribution of data affects performances. With HPF, programmers provide a few directives to specify processor and data layouts, and express parallelism by means of parallel loop constructs or collective operations on arrays [23], e.g. collective Fortran 90 operators. The compiler manages data distribution and, if the target parallel architecture is distributed–memory, translates the source program into an SPMD code with explicit interprocessor communications and synchronizations. The view that programmers have of the target machine is sufficiently high-level, and the performances of the codes obtained by compiling *regular* applications are often comparable to those obtained with the hand–coded optimized versions of the same problems.

Irregular problems, on the other hand, are a big challenge for data–parallel languages, compilers and run-time support designers. The main problem concerns the impossibility for programmers and compilers to be aware before run-time of some features of computation that may significantly affect the final performance. These features are, above all, irregular communications and load imbalances. For example, several data–parallel scientific applications make use of indirectly accessed arrays. The compilers cannot implement these data references by means of regular communications, and often cannot restructure programs to allow computations to overlap with communications. Load imbalance, as we mentioned above, is another feature of a data parallel computation that may worsen the performance when it cannot be detected until run–time.

In this paper we consider a particular class of irregular problems, *non–uniform* data–parallel computations which exploit regular and predictable data references. These computations are usually expressed by means of parallel loop constructs characterized by iteration execution costs which are variable and often unpredictable (*non–uniform parallel loops*). On the other hand, the data referenced by each loop iteration are known at compile–time so that optimizations to minimize overheads and hide the latencies of communications needed to implement remote data retrieval can be applied. To efficiently implement non–uniform parallel loops on *distributed–memory multiprocessors*, the *binding* of data to memories, and of computations to processing elements, must be solved in a way that maximizes locality exploitation and, at the same time, minimizes loop completion time by achieving balanced processor workloads. The binding may be *static*, i.e. established before running the program, *dynamic*, i.e. postponed at run-time, or *hybrid*, i.e. when a combination of both forms of binding is chosen, so that while an initial binding is statically chosen, it can be dynamically modified at run-time to obtain better performances.

- A completely static binding is followed by current HPF compilers to reduce interprocessor communications and related overheads. Data distribution is suggested by programmers, while the compiler derives the mapping of the iterations to the various processing elements according to a static rule, e.g. the *owner computes rule*. To specify data mapping, programmers can only choose between regular *block* and *cyclic* distributions, or a combination of both. The block distribution assigns contiguous array elements to the local memory of the same processor, and thus generally allows data locality to be exploited. Conversely, it may produce highly unbalanced workloads if contiguous elements have similar costs. To achieve a better load balancing, a cyclic distribution for the arrays involved can be adopted, which scatters data elements among the local memories of the various processors. Finally, a *cyclic*(k) distribution, which cyclically assigns contiguous blocks of k elements, may be chosen to mitigate the poor locality of pure cyclic data layouts, and to tradeoff between data locality and load balancing.
  Data layout can also be changed at run–time by declaring an array as `DYNAMIC`, and by inserting an executable redistribution directive into the code, which allows a different regular distribution to be forced for the array at that point which is more suitable for performing a given calculation. This HPF feature can be exploited, as we will show later, to turn a block–distributed array, accessed during a uniform parallel loop involving stencil data references, into a cyclic layout before executing successive non–uniform computations.

  Several extensions have been proposed for the HPF language to support irregular distributions [4], and some of these have been included in version 2.0 of the standard [5]. With an accurate use of DYNAMIC attributes and GEN_BLOCK irregular redistributions, an expert HPF-2 programmer can in many cases meet both locality exploitation and load balancing goals. The computation of a good irregular distribution is however left to the user who, to obtain high efficiencies, must give up most of the advantages of using a high–level parallel language.

- In dynamic approaches the actual binding of computations and data takes place at run-time. Run–time binding is commonly exploited on Uniform Memory Access (UMA) shared–memory multiprocessors. All the loop iterations are stored in a shared queue, and each processor self schedules them by accessing the central queue. The binding of data to the local memory of each processor, i.e. the memory–coherent local cache, is in this case implicit, and occurs at run-time when a given data element is actually accessed. Many *Self Scheduling* policies have been proposed that are aimed at reducing synchronizations and contention overheads while achieving a good load balance [18, 24, 8]. Recently, however, it has been proved that, also on UMA parallel architectures, dynamic iteration assignment should be guided not only by considering the load balance goal, but also data reuse and locality exploitation. Markatos and LeBlanc [12] have investigated a scheduling strategy, based on a static partitioning of iterations, which initially assigns iterations for *affinity* with previously assigned ones. Thus, also for shared–memory machines, some forms of hybrid (static+dynamic) binding of data and computation have proved to be valuable to attain a better performance.

  Similar dynamic approaches can also be adopted on distributed–memory, message passing machines. In this case, a paradigm, known as *task queue* (or *task farm*) is employed. At run–time a centralized dispatcher distributes loop iterations and related data, and a set of workers self–schedule them by asking the dispatcher for further work. Hierarchical or distributed schema can be exploited to avoid the introduction of a single centralization point [11], but in any case such completely dynamic approaches introduce considerable overheads and impose strong storage constraints on the size of the addressable problems. Moreover, when the considered non–uniform parallel loop has to be executed several times on the data produced by the previous execution, all data should be returned to the dispatcher before starting the next execution, thus introducing even larger overheads.

- Finally, in hybrid binding approaches data and computations are first assigned at compile time, but this initial assignment can be dynamically changed at run-time to mitigate load imbalance. The binding can change at fixed instants, i.e. after a global synchronization, or asynchronously on the basis of the decisions taken by the dynamic scheduling policy used.

  In the former case, synchronizations are fixed a priori to occur at given points within the parallel program, e.g. between two consecutive executions of the same unbalanced loop. Workload redistribution decisions are taken on the basis of past knowledge only, and thus possible changes in the workload distribution among successive executions might affect the efficiency of the method. Moreover, processors at least incur the imbalance deriving from the first execution of the loop. Another disadvantage of this technique is the impossibility to overlap communication/synchronization phases with useful computations. In fact, computation restarts after the synchronization phase that may redistribute processor workloads. On the other hand, synchronization points can be used to make a global knowledge of workload distribution available to each processor. This global knowledge can be exploited to minimize the number and volume of communications needed to redistribute the workload, thus making the technique particularly suitable for loosely coupled systems, where communication costs are high.

  In the latter case, an asynchronous dynamic scheduling strategy is employed, which migrates computation and data on the basis of a local knowledge of actual workloads, and does not introduce synchronization points. To some extent, this approach can be considered as a completely distributed implementation of the centralized task queue mentioned before. Each processor is simultaneously a dispatcher with respect to the local queue of those computations statically assigned to it, and a worker, which self–schedules computations belonging either to its local queue or to queues managed by other processors. Remote queues of overloaded processors are accessed only when a processor detects that its local queue is becoming empty. If we still have enough computations to carry out, a careful prefetching technique can be exploited to hide communication latencies. The disadvantage of such techniques is the need to employ several short messages to implement dynamic policies in a completely distributed way. If the communication subsystem of the target parallel architecture is not efficient, and/or communication overheads cannot be overlapped with computation because the size of the problem or the computational grain are too low, the actual efficiency may be degraded.

In the rest of the paper we will compare static and hybrid implementation strategies which can be used to balance the workloads derived from the execution of non–uniform parallel loops. Due to the high overheads involved, we will not consider completely dynamic self–scheduling approaches. The application we consider as a benchmark is a two–dimensional flame simulation code, a code where two parallel loops that access the same arrays must be executed several times to carry out consecutive steps of the simulation. While the former loop is uniform and exploits stencil data references, the latter one is unbalanced since the time it takes for processing each point of the simulation grid varies considerably and may change as the simulation progresses. We will show that an asynchronous hybrid technique like the one outlined above is the best one when employed to implement the flame simulation benchmark on a Cray T3E, an architecture that offers a very efficient support for communications. The experiments were conducted by using synthetic input data sets that introduce a known level of load imbalance.

The paper is organized as follows. Section 2 describes the kernel of the two–dimensional flame simulation benchmark, while Section 3 presents the static and hybrid approaches and gives some details on their implementation. The load imbalance model used to build the synthetic data sets used for the experiments is described in Section 4, which also presents the assumptions made to carry out the tests. The experimental results are reported and discussed in depth in Section 5. Finally, Section 6 draws the conclusions.

2 The benchmark

The benchmark we used to evaluate the various approaches outlined in Section 1, is directly derived from a class of scientific codes used to carry out detailed flame simulations [16]. Flame simulations have attracted growing interest in the last few years: highly detailed simulations can in fact give scientists important information on fuel flammability limits and on the interactions between chemical and physical processes. A detailed study of the structure, stability and dynamics of a variety of flames and fires is accomplished by simulating fluid dynamics, chemical reaction kinetics, diffusive transport of species, radiation, and other heat losses. For heavier hydrocarbons, the computational requirements of detailed flame simulations are already beyond the capabilities of traditional supercomputers, thus massive parallelism has to be used. One example of code that performs a time-dependent multi-dimensional simulation of hydrocarbon flames is described in [19], and is included in the HPF–2 draft [4] as one of its motivating application codes. The HPF-like code of the kernel of a simplified two-dimensional flame modeling program is reported in Figure 1. The simulation is split into two distinct computational phases executed at each time step. The first phase computes fluid *Convection* over a structured mesh. The computation at each point has similar costs, and involves the point itself and the nearest neighboring elements of the simulation mesh. Stencil communications are thus required at each time step to carry out this phase on a distributed memory machine. The second phase simulates chemical *Reaction* and subsequent energy releases represented by ordinary differential equations. The solution at each grid point only depends on the value of the point itself, but its computational costs may vary significantly since the chemical combustion proceeds at different rates across the simulation space. The Reaction phase globally requires more than half of the total execution time, and most of this time is spent on a small fraction of the mesh points. Furthermore, the computation is adaptive since the workload distribution evolves as the simulation progresses.

The most interesting characteristic of flame simulation codes is that, in order to efficiently exploit a highly parallel machine, two conflicting requirements have to be dealt with: the need to adopt a regular block partitioning to take advantage of data locality during the first Convection phase, and, conversely, the need to balance the highly variable workloads during the Reaction phase. This second point would entail a different distribution of the simulation grid on the processor's local memories. Unfortunately, since the workload distribution evolves and depends on the specific features of the input data set, at compile–time we cannot devise a data distribution that is suitable for both phases of the simulation.

```
C  Loop on the time steps
   DO k= 1,K
C    Convection phase over a structured mesh
C    Nearest neighbor communications - Constant computation time
     FORALL (i= 2:N1-1,  j= 2:N2-1)
       A(i,j) = A(i,j) + F(B(i,j), B(i-1,j), B(i,j-1), B(i+1,j), B(i,j+1), C(i,j))
     END FORALL
     B = A
C    Chemical reaction phase - No communications - High load imbalance
C    Workload distribution evolves during simulation
     FORALL (i= 1:N1,  j= 1:N2)
       C(i,j) = Reaction(A(i,j))
     END FORALL
   END DO
```

Figure 1: HPF-like code of a simplified two-dimensional flame modeling code.

3 Benchmark implementations

This section discusses four possible implementations of the flame simulation code illustrated above. In particular, we compare two static and two hybrid solutions for balancing the non–uniform workloads deriving from the execution of the Reaction phase of the simulation:

- the first static solution adopts a regular cyclic data layout for both dimensions of the arrays involved. Data locality, which should be exploited by the first Convection loop, is thus sacrificed to the load balancing needs of the second Reaction loop;
- the second static approach uses a block distribution during the first loop, but the layout of the simulation grid is turned to cyclic before executing the second Reaction loop. The block distribution must be restored at the end of each time step. We still define this approach as static because the *binding* of data and computations is still decided at compile-time, although re-distribution occurs at given instants during execution;
- the first hybrid approach is based on the Inspector-Executor paradigm [2]. It is in some sense similar to the previous static approach, because a block distribution is used during the first Convection loop, and redistribution takes place at fixed points during the computation. However, the actual binding of data and computation is decided dynamically, on the basis of information collected at run-time during a previous simulation time step. This means that, the first time a given unbalanced loop is executed, the distribution is the statically chosen block one, since no information has yet been collected;
- the second hybrid approach exploits the SUPPLE support [13, 14, 15], which always adopts a statically fixed block distribution. During execution, chunks of iterations and associated data may be dynamically moved toward underloaded processors. In this case both the time instants when a possible migration may occur, and the actual migration plan, are decided dynamically at run-time.

We implemented all the above solutions on a Cray T3E exploiting the same message–passing layer (MPI) and, where possible, applying the same optimization techniques. This allowed us to evaluate and compare on the same basis the effectiveness of the implementations, since the experimental results are not invalidated by incomparable factors introduced by the use of different tools/mechanisms/optimizations. We think that the problems we have faced to design the flame simulation benchmark, in particular load balancing, locality exploitation and management of stencil communications, are common to many irregular applications. Thus the following discussion, although a specific case study, can be considered as general.

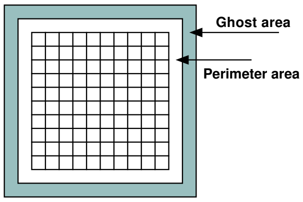


Figure 2: Ghost, perimeter and inner areas of a block partition.

3.1 Implementation of the Convection phase

Before detailing the solutions adopted for the non–uniform Reaction phase, it is worth discussing the optimization techniques used to exploit the regularity of the parallel loop implementing the Convection phase of the flame simulation. This loop is uniform and accesses data with a regular and constant *Five-Point stencil*. The techniques used to implement these stencil references depend on the layout adopted to distribute the data. The last three solutions discussed in this paper use a static block data layout and a static scheduling of iterations for its implementation, while the first one exploits a cyclic data layout for both parallel loops.

Block data layout improves locality since it reduces the number of accesses to remote data needed to implement stencil references. According to this data layout, arrays `A`, `B` and `C` are aligned to each other and distributed in blocks of equal size on the processor grid. For the array `B`, which is accessed by stencil data references, we allocate on each processor enough memory to host the block partition, logically subdivided into an *inner* and a *perimeter* region, and a surrounding *ghost* area (see Figure 2). The *ghost* area is used to buffer the parts of the *perimeter* areas of the adjacent partitions which are owned by neighboring processors and are accessed through non local references. An equal fraction of loop iterations is statically assigned to each processor according to the owner computes rule, and the loops executed by each processor are *localized*, i.e. their boundaries and array references become relative to the local array block. Before executing the loop, however, non local data accessed by stencil references must be fetched from the nearest neighboring nodes. This means that explicit communications must be inserted in the code. Several scheduling and communication optimizations can be exploited to this purpose to reduce overheads and hide communication latencies [7, 20]. Overhead minimization is accomplished by avoiding sending several messages or replicated data to the same processor, i.e. by exploiting message vectorization, coalescing, and aggregation optimizations [7]. Moreover, to overlap communications with useful computations, iterations are reordered [7]: the iterations that assign data items belonging to the inner area (which refer local data only) are scheduled between the asynchronous sending of the perimeter areas to neighboring processors and the receiving of the corresponding data into the ghost areas.

The execution of the iterations that assign the elements lying on the inner area was also optimized. We grouped these iterations into *chunks* of a fixed size. To this end, each inner block partition is statically *tiled* (Figure 2 shows the tiling of the inner area, which appears to be subdivided into further small sub-blocks). Tiling is a useful optimization which improves locality exploitation [10], and is also exploited in the non–uniform Reaction loop. In fact to reduce overheads, in both the hybrid approaches mentioned above, chunks and associated data instead of single iterations are migrated on the basis of the workload distribution.

Finally, since the hybrid approaches may migrate data tiles, but the results of remote executions must be returned to the owner of each tile, some forms of synchronization must be used before reusing data, to avoid accessing non coherent data. This is the case of the array `C`, which is modified by the second non–uniform Reaction loop, and which is read by the first Convection loop. Since tiles of `C`, due to dynamic work migration, may be updated by processors that are not the owners, we associate a *full/empty-like* flag [1] with each data tile. If a chunk that modifies a given tile is migrated, then the associated flag is set. The flag will be unset when the updated data tile is received back from the remote processors. This solution avoids the introduction of a barrier synchronization at the end of the Reaction loop to wait for coherence to be restored. We ensure coherence by checking flags associated with each tile to be read, thus allowing useful computations to be overlapped with communications.

The only solution where we used a different technique to implement the Convection loop is the one where a fixed cyclic data layout is adopted for both dimensions of the arrays involved. In this case too, the scheduling is static and derived from the owner computes rule. No iteration reordering is possible, however, to overlap computation with the communication time to receive remote data. In fact, due to the stencil data references, for each assignment that is executed by a generic processor p, and that reads a five point stencil of the array `B`, we can easily see that all the points but the center of the stencil are statically allocated to processors that are different from p. In particular, because in all our experiments we implemented the benchmark on a logical two–dimensional processor grid, all remote points are owned by the four neighboring processors of p.
Instead of inserting explicit communications for remote data retrieval inside the parallel loop, a simple static analysis of the source program allows one to determine that message *vectorization* and *aggregation* optimizations [7] can be applied to minimize communication overheads. As a consequence of these optimizations, at each time step, before executing the Convection phase each processor $p_{h,k}$, with position (h, k) in the two–dimensional grid, sends its partition of the array `B` to the four neighboring processors $p_{h-1,k}$, $p_{h+1,k}$, $p_{h,k-1}$, $p_{h,k+1}$, and correspondingly receives from them a copy of their partitions of `B`. At this point, each processor $p_{h,k}$ can carry out the Convection phase by transforming each remote stencil reference into a reference to the locally buffered copy of the corresponding adjacent partition. For example, the remote reference `B(i-1,j)` performed by processor $p_{h,k}$ is transformed into a local reference to $\mathtt{B}_{h-1,k}$`(i,j))`, where $\mathtt{B}_{h-1,k}$ is the partition of `B` that has been statically assigned to $p_{h-1,k}$ according to the cyclic distribution.

3.2 Implementation of the Reaction phase

In this section we detail and compare the solutions adopted to implement the irregular part of our flame simulation benchmark: the non–uniform parallel loop performing chemical reaction.

Cyclic data layout

The simplest way to achieve a good workload balance among the processors used to carry out the flame simulation, is to give up the idea of exploiting data locality by scattering both dimensions of the simulation grid among the local memories of the various processors. This solution allows one to adopt a simple inexpensive static policy for iteration scheduling based on the *owner computes rule*, and to obtain in most cases almost equal completion times for the parallel loop that implements the Reaction phase of the simulation.

Dynamic redistribution

This solution tries to take full advantage of the possible static policies that can be adopted to implement both the loops of the benchmark. A block layout is adopted for the first, regular Convection phase, and a cyclic distribution is exploited for the non–uniform Reaction loop. Hence, an appropriate code performing block–to–cyclic (cyclic–to–block) dynamic redistribution must be inserted at the end of the first (second) parallel loop.

Dynamic redistribution requires communications involving all the processors. Each processor has to send data to all the others, so that the same processor has to receive data from all the others. To reduce the number of messages, data for transmission to the same processor are *aggregated* [7], so that each processor sends only one message to each other processor. The order in which messages are sent differs from processor to processor, thus preventing several processors from simultaneously sending messages to the same destination, which would cause network *contention* problems.

The Inspector-Executor approach

The solution to many problems whose irregularities prevent compile-time optimizations, can be addressed by exploiting *Inspector-Executor* codes which collect information at run-time, and then use this information to optimize subsequent computations [2]. The CHAOS/PARTI libraries [21] developed by the group headed by Joel Saltz at the University of Maryland, and adopted by the Vienna Fortran Compilation System [26] and by other HPF-like compilers, are the most notable examples of this approach. CHAOS/PARTI functionalities provide run–time support for irregular array distributions, loop scheduling based upon the *almost owner computes* rule, data redistribution, and data accessed through indirection arrays.

Non–uniform parallel loops are supported as well: information about varying iteration execution costs is collected at run-time, and is consequently used to build a *workload migration plan*. Such a *workload migration plan* is exploited by the overloaded processors to migrate work in excess towards underloaded processors.

If we refer to the code in Figure 1, we can outline how an *Inspector-Executor* implementation approach can be exploited in this case. This solution was inspired by [22, 19] which describe how the CHAOS/PARTI run–time libraries can be exploited to implement the benchmark discussed in Section 2. It allows us to satisfy the two contrasting requirements discussed in Section 2, since the simulation grid is block-distributed to enhance the locality of references during the first Convection phase of the simulation, and part of the iterations are dynamically scheduled during the Reaction phase to achieve load balancing.

1. During each iteration k of the outer loop, processors profile the Reaction loop execution and store iteration costs on *local_costs*, a array with dimension $N1 \times N2$. Once the non–uniform loop has finished, each processor p_i knows the costs of each iteration of the Reaction loop, and its own total load $TL(p_i)$. This total load is thus broadcast to all the other processors.

2. On the basis of the known load on the other processors, each processor locally decides if balancing actions have to be taken. A simple heuristics which measures the difference among the loads of the most heavily and lightly loaded processors, and compares it with a fixed *threshold* was used for this purpose. If a load imbalance is detected, all the processors execute a *binpack* algorithm [22] to build the same suboptimal *workload migration plan* which states how much work should be moved from each overloaded processor to each of the underloaded processors to achieve load balance. Suppose that $\overline{M}$ is the average processor load, and there are m, $1 \leq m < P$, overloaded processors $p'_0, p'_1, \cdots, p'_{m-1}$, and n, $1 \leq n \leq (P-m)$ underloaded processors $\overline{p}_0, \overline{p}_1, \cdots, \overline{p}_{n-1}$. The algorithm, for each overloaded processor p'_i, $1 \leq i < m$, generates a list of load amounts $\alpha^i_0, \alpha^i_1, \cdots, \alpha^i_{n-1}$ where $\alpha^i_j \geq 0$ is the amount of load that should be moved from p'_i to $\overline{p}_j$. To effectively balance the processor workloads, amounts α^i_j have to be chosen so that, for all overloaded processors p'_i, $\sum_{j=0}^{n-1} \alpha^i_j \approx TL(p'_i) - \overline{M}$, and, for all underloaded processors $\overline{p}_j$, $\sum_{i=0}^{m-1} \alpha^i_j \approx \overline{M} - TL(\overline{p}_j)$. This is achieved by taking each overloaded processor p'_i, and by moving part of its excess workload $(TL(p'_i) - \overline{M})$ to an underloaded processor $\overline{p}_j$ until $\overline{p}_j$'s workload becomes approximately equal to $\overline{M}$. After this workload migration, if p'_i's workload is still greater than $\overline{M}$, another underloaded processor is chosen and the process is repeated. Conversely, if p'_i's workload is not sufficient to balance processor $\overline{p}_j$'s load, $\overline{p}_j$ will receive other work from another overloaded processor. To reduce the number of messages to be exchanged, a heuristics which chooses the most overloaded and most underloaded processors first is used to select the source and destination of each workload migration.

3. At the end of the previous step, overloaded processors know exactly how much of their load should be transferred to each underloaded processor. Assuming that the workload distribution over the simulation grid changes slowly, they can profitably use the *workload migration plan* to try to balance the load during the next iteration of the outer loop. However, n bunches of iterations $\beta^i_0, \beta^i_1, \cdots, \beta^i_{n-1}$ with costs $\alpha^i_0, \alpha^i_1, \cdots, \alpha^i_{n-1}$ respectively, have to be chosen first by each overloaded processor p'_i for sending. To reduce the communication volume the dimensions of the bunches have to be minimized by choosing the heavier iterations for migration. Thus, p'_i sorts iteration indexes on the basis of the costs stored in array *local_costs*, and begins putting the heavier iterations in bunches β^i_j, $j = 0, 1, \cdots, n-1$, until amounts α^i_j are reached. At the end of this step, bunch β^i_j is empty if no iteration has to be moved from p'_i to $\overline{p}_j$ according to the established *workload migration plan* ($\alpha^i_j = 0$). Elsewhere, it contains iterations whose total cost is approximately equal to α^i_j. The resulting list of n couples $(\overline{p}_j, \beta^i_j)$ constitutes the *communication schedule* of each overloaded processor p'_i.

4. At iteration $k+1$, overloaded processors use their *communication schedule* to balance the workloads that are derived from the execution of the non–uniform loop. To this end, after the Convection phase of the simulation, but before starting the Reaction phase, each processor p'_i sends each bunch β^i_j,

$j = 0, 1, \cdots, n-1$, to the respective processor $\overline{p}_j$. Then p'_i performs non-migrated, local iterations and in return receives the results of migrated bunches. Correspondingly, once both the Convection and Reaction phases of the simulation have been executed on the owned block-partition, underloaded processors receive the bunches of migrated iterations from the overloaded processors. They then execute the iterations of each bunch, and send back the results.

Since the computational costs associated with the iterations of the Reaction loop may vary as execution progresses, processors must continue profiling the non–uniform parallel loop during all the simulation. Each outer loop iteration thus requires a collective communication to broadcast the current processor loads (*all_gather* operation). Only after the *all_gather* operation, can processors locally decide if the workload is still unbalanced, and, if so, they build a new *workload migration plan* and *communication schedule*.

This approach is very different from the previous one that is based upon the complete redistribution of the simulation grid. In this case, in fact, a global knowledge of processor loads is exploited to decide data migration and minimize data and computation reallocation. On the other hand, this minimization of the communication volume leads to the need to profile at run–time the execution of every non–uniform loop, and with the cost of updating the global knowledge on the current workload distribution for each processor. In addition, the cost of building both the *workload migration plan* and *communication schedule* every time a load imbalance is detected, is not negligible. To reduce this cost an $O(n)$ algorithm for the approximate sorting of iterations is proposed in [22] for step 3 of the above implementation scheme. However, we followed a different direction. In our implementation, whose results are reported in Section 5, we treat a chunk of iterations, rather than a single iteration, as the scheduling unit. This allows us to reduce considerably not only the cost of sorting but also the costs related to loop execution profiling and *communication schedule* building.

To conclude our discussion of the *Inspector-Executor* solution, it is worth noting that, unlike all the other approaches discussed in this paper, an *Inspector-Executor* paradigm can be profitably exploited only if the non–uniform loop is iterated many times and the workload distribution does not change, or changes very slowly, as the execution progresses. The first execution of each loop is by necessity accomplished according to a static schedule, thus jeopardizing part of the benefits of subsequent dynamic load balancing actions if the workload is very unbalanced and the parallel loop is executed only a few times.

The SUPPLE approach

SUPPLE is a run-time support for the implementation of both uniform and non–uniform parallel loops [13, 14, 15]. It only supports block-distributed arrays to exploit the locality that derive from regular *stencil* references. The innovative feature of SUPPLE is its efficient support for non–uniform loops such as the Reaction loop in the flame simulation code. For this purpose, SUPPLE exploits an innovative hybrid (static + dynamic) scheduling technique which adopts local policies of imbalance detection to avoid global synchronizations, and hides most dynamic scheduling overheads by overlapping communication with useful computations. Differently from other proposals [17], SUPPLE scheduling policy is fully distributed and does not introduce bottlenecks which may jeopardize the efficiency when many processors are used.

At the beginning, to reduce overheads, iterations of the Reaction loop are scheduled statically, by sequentially executing the chunks stored in a queue Q, hereafter *local* queue. Once a processor understands that its local queue is *becoming empty*, the dynamic part of the scheduling policy begins. It tries to balance the processor workloads by migrating both chunks and corresponding data tiles from overloaded to underloaded processors. Migrated chunks and data tiles are stored by each receiving processor in a queue RQ, called *remote*. The dynamic technique can be classified as a *receiver initiated* load balancing technique, since it is the *receiver* of remote chunks that asks overloaded processors for work migration [9, 25].

The SUPPLE scheduling algorithm is fully distributed and asynchronous. Chunk migration decisions are taken on the basis of local information only as follows:

- During the initial static phase, each processor only executes local chunks belonging to Q. It can thus measure chunk costs and use the average cost to estimate its workload. Moreover, when chunks are migrated toward underloaded processors, an estimate of their cost is communicated as well. Processors can thus estimate their own *current load* at any time by inspecting their queues Q and RQ.

- When the estimated local load becomes lower than a machine-dependent *Threshold*, each processor autonomously starts the dynamic part of the scheduling technique and begins asking other processors for remote chunks. Note that the *Threshold* parameter is chosen high enough to prefetch remote chunks, thus preventing underloaded processors from becoming idle while waiting for chunk migrations. The processors that are asked for chunk migration are chosen on the basis of an inexpensive *round-robin* policy. As discussed by Kumar et al. [9], it is in fact more important not to spend too much time in making a decision than always making the best decision. Correspondingly, the processors which receive a chunk migration request decide whether or not it should be granted by comparing their *current load* with the same *Threshold*. In fact a processor p_j, which receives the migration request from a processor p_i, will grant the request by moving some of its workload to p_j only if its *current load* is higher than *Threshold*.
  To reduce the overheads which might derive from requests for remote chunks which cannot be served because the *current load* of the processor that has been asked for is too low (i.e., it is lower than *Threshold*), each processor, when its *current load* becomes lower than *Threshold*, broadcasts a so-called *termination message*. These messages are used by the receiving processors to update their local knowledge of the global status of the computation. The round-robin strategy used by underloaded processors thus selects source processors only from those that have not yet communicated their termination.

- Once an overloaded processor decides to grant a migration request, it must choose the most appropriate number of chunks that must be sent to the asking processor. SUPPLE uses a modified *Factoring* scheme to determine this number [8]. *Factoring* is a *Self Scheduling* heuristics formerly proposed to address the efficient implementation of parallel loops on shared–memory multiprocessors. It provides a way to determine the appropriate number of iterations that each processor must fetch at each access from a central queue which stores the indexes of unscheduled loop iterations. Clearly, the larger this number is, the lower the contention overheads for accessing the shared queue, and the higher the probability of introducing load imbalances when the fetched iterations are actually executed. *Factoring* requires that, if u is the number of remaining unscheduled iterations and P is the number of processors involved, the next P requests for new work are granted with a bunch of $\frac{u}{2 \cdot P}$ iterations. Consequently, at the beginning large bunches of iterations are scheduled, and this number is reduced when the shared queue is going to be emptied. In our case, instead of having a single queue, we have multiple shared queues, one for each processor. When a loop is unbalanced only some of these queues, i.e. those owned by overloaded processors, will be involved in granting remote requests for further work. The other difference regards our scheduling unit, which is a chunk of iterations instead of a single iteration. The modified heuristic we used is thus the following: an overloaded processor replies to a request for further work by sending $\frac{k}{2 \cdot P}$ chunks, where k is the number of chunks currently stored in Q, and P is the number of processors.

Moreover, the policies exploited by SUPPLE to manage data coherence and termination detection are fully distributed and asynchronous. The *full/empty-like* technique [1] described above is used to asynchronously manage the coherence of migrated data tiles. When processor p_i sends a chunk b to p_j, it sets a flag marking the data tiles associated with b as invalid. The next time p_i needs to access the same tiles, p_i checks the flag and, if the flag is still set, waits for the updated data tiles from node p_j. Such a control of data consistency does not entail waiting for coherence messages at the end of parallel loop execution: a new loop can be started and useful computations can be overlapped with these communications. As far as termination detection is concerned, the role of a processor in the parallel loop execution finishes when it has already received a termination message from all the other processors, and both its queues Q and RQ are empty.

4 The model of load imbalance

To experimentally evaluate the behavior of the implementation strategies discussed in the previous sections, we built a set of synthetic input data sets for the flame simulation benchmark whose skeleton is shown in Figure 1. The synthetic data sets are 1024×1024 arrays which represent possible states of the simulation grids which result in different workload distributions. The per–point time needed to execute function `Reaction()` of the flame simulation kernel has been forced to be directly proportional to the value of the corresponding data set point.

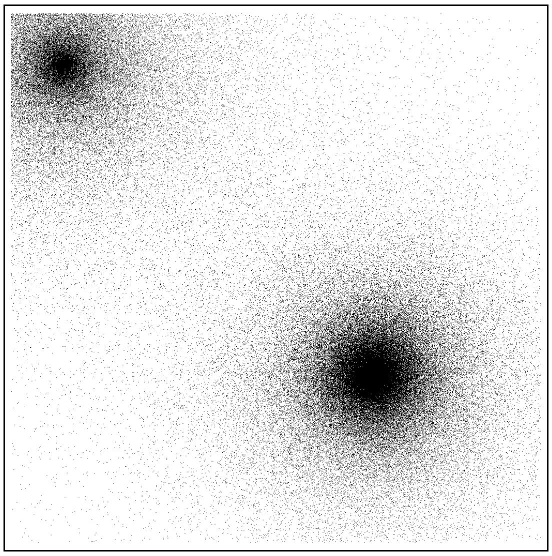


Figure 3: Representation of a synthetic data set.

Given μ, the per–point execution time averaged over all the points of the simulation grid, the synthetic data sets were built according to a model of load imbalance that assumes that the workload is not distributed uniformly, and that the computational cost of a given point may be *high* (i.e. greater than μ) or *low* (i.e. lower than μ). Moreover the model assumes that d, $0 < d < 1$, is the fraction of simulation grid points with *high* costs, while t, $0 < t \leq 1$, is the fraction of the total workload T equally distributed among the *high*–cost points. Clearly, since the time needed to process *high*–cost points is greater than μ, it follows that $t \geq d$. In the choice of the positions for the *high*–cost points we followed an exponential distribution with respect to two points c_1 and c_2 placed on the simulation grid. As a result, about two thirds of the *high*–cost points are concentrated in a region whose center is c_1, while the other third of the points are concentrated near c_2. Figure 3 shows a representation of a data set characterized by $d = 0.1$ in which *high*–cost (*low*–cost) points are displayed as black (white). Note that the parameter d determines the dimension of the loaded regions. Since larger values of t correspond to larger fractions of T concentrated on the two loaded regions, the workload imbalance is directly proportional to t for a given value of d. Similarly, the imbalance is inversely proportional to d for a given value of t. From these remarks, we can derive F, called the *factor of imbalance*, which is defined as

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

In the tests, we employed data sets characterized by values of F ranging from 1 to 9, where the fraction of loaded points is kept fixed ($d = 0.1$). Note that the case $F = 1$ corresponds to a data set that does not introduce any workload imbalance, since the computational cost associated with each grid point is exactly equal to μ.

5 Experimental results

All the benchmark implementations were coded in C by exploiting MPI as a message–passing layer. The tests were conducted on 64 Cray–T3E nodes arranged as a logical 8×8 processor grid. The same 1024×1024 synthetic data sets were fed in input to the benchmark versions so that a partition of 128×128 points is assigned to each processing nodes. The points assigned to each processor are contiguous if a block distribution is exploited, and *scattered*, with a constant stride of eight in both dimensions, when a cyclic data layout is adopted instead.

Table I: Reaction phase: per–point execution time for some values of μ and F

μ	F	*high*–cost points	*low*–cost points
0.038 $msec$	2	0.075 $msec$	0.033 $msec$
0.3 $msec$	2	0.6 $msec$	0.267 $msec$
0.038 $msec$	5	0.187 $msec$	0.021 $msec$
0.3 $msec$	5	1.498 $msec$	0.167 $msec$
0.038 $msec$	9	0.337 $msec$	0.004 $msec$
0.3 $msec$	9	2.697 $msec$	0.033 $msec$

Table II: Optimal Completion Time (OCT) on 64 processors for different values of μ

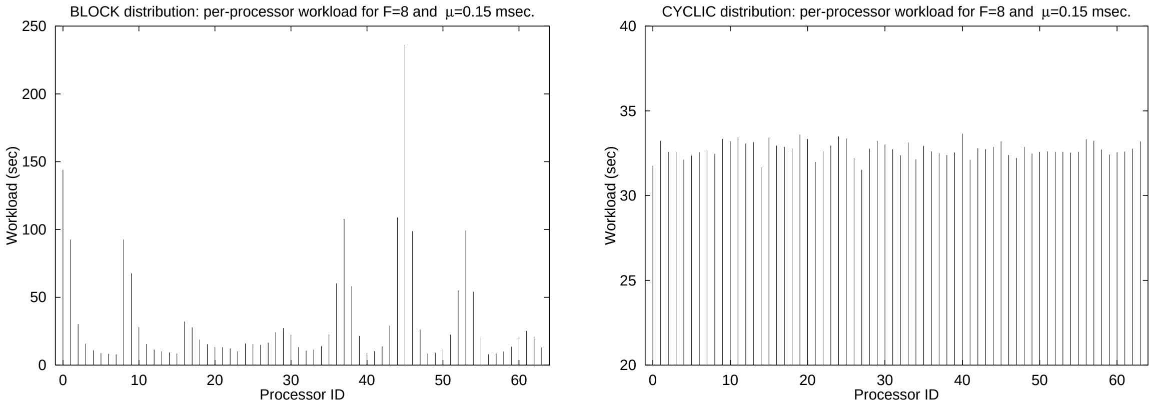
μ	0.038 $msec$	0.075 $msec$	0.15 $msec$	0.3 $msec$
OCT	8.19 sec	16.38 sec	32.76 sec	65.53 sec

We assumed that one quarter of the total execution time is spent in the Convection phase of the flame simulation benchmark, while the Reaction phase takes the remaining three quarters. Hence, if μ is the average execution time to process a grid point during the Reaction phase, the average execution time per point during the Convection phase is thus $\frac{\mu}{3}$. We conducted experiments with input data sets characterized by several values of μ and, specifically, for $\mu \in \{0.038\ msec,\ 0.075\ msec,\ 0.15\ msec,\ 0.3\ msec\}$. The external sequential loop of the flame simulation code was iterated for 10 time steps. Table I reports some examples regarding the imbalance introduced by our synthetic data sets. In particular, it shows the times needed to compute each point during the Reaction phase, and distinguishes between *high* and *low*–cost points. Table II shows, for various values of μ, the theoretical Optimal Completion Time (OCT) when the simulation code is run on 64 processors. This time, which does not take into account any overheads, is computed by multiplying the average time required to process a single point of the simulation grid ($\mu + \mu/3$), times the number of points assigned to each processor (128×128), times the number of simulation time steps (10).

Figure 4 shows the per–processor workloads that derive from *block* (Figure 4.(a) and *cyclic* (Figure 4.(b) distributions of the simulation grid. The workload is expressed in seconds, and is relative to the execution of 10 time steps of the simulation on the data set characterized by $F = 8$ and $\mu = 0.15$ msec. (OCT equal to 32.76 seconds). As expected, the *block* data distribution results in a huge load imbalance if a static scheduling policy is exploited: the processor workloads for the case considered range from 8 to 236 seconds. Acceptable, even if not optimal, the workload distribution that derives from the *cyclic* data layout. Processor workloads range, in this case, from 31.52 to 33.65 seconds.

Figures 5 to 8 report the results of the experiments conducted to evaluate the four implementation strategies that we have discussed for the flame simulation benchmark. Each plot regards a different implementation and shows several curves related to distinct values of μ, the average iteration execution time of the Reaction parallel loop. For each $\mu \in \{0.038\ msec,\ 0.075\ msec,\ 0.15\ msec,\ 0.3\ msec\}$ and $1 \leq F \leq 9$, the curves plot the difference between the completion time of the slowest processor and the theoretical OCT (see Table II). This difference can be considered as the sum of all overheads due to communications, residual workload imbalance, and loop housekeeping. It therefore also includes the time to send/receive messages that implement the stencil communications of the first convection loop. As can be seen, all the implementations introduce a low overhead even in the balanced case ($F = 1$), and the overhead tends to increase as data sets characterized by more unbalanced load distributions are processed.

In the case of the CYCLIC implementation the differences from the OCT shown in Figure 5 range from 0.20 to 0.37 seconds for $\mu = 0.038\ msec.$, and from 0.21 to 1.63 seconds for $\mu = 0.3\ msec$. On the one hand, these results show that the T3E communication subsystem is very efficient, and that the optimization techniques really reduce the overheads of the communications needed to implement the stencil references of the Convection parallel loop. The adoption of a cyclic distribution of the simulation grid requires, in fact, that each processor sends at each time step its data partition to the four neighboring processors, and



(a) (b)

Figure 4: Per–processor workload for *block* (a) and *cyclic* (b) distributions of the 1024×1024 simulation grid on the 8×8 processor grid: case $F = 8$ and $\mu = 0.15$ msec.

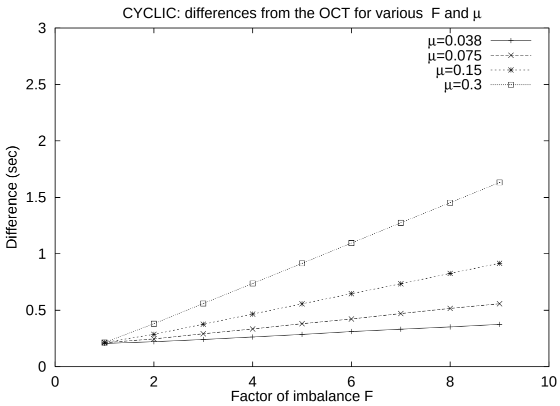


Figure 5: CYCLIC implementation: differences from the OCT for various values of μ as a function of the factor of imbalance F.

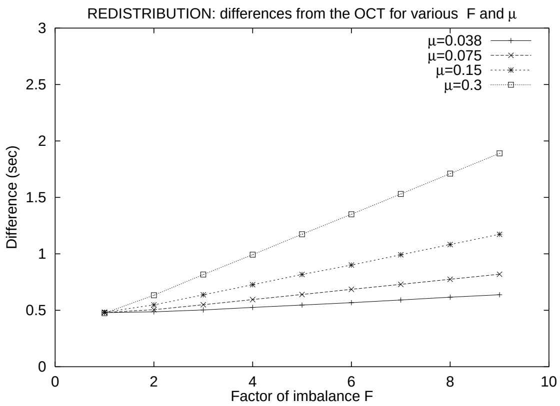
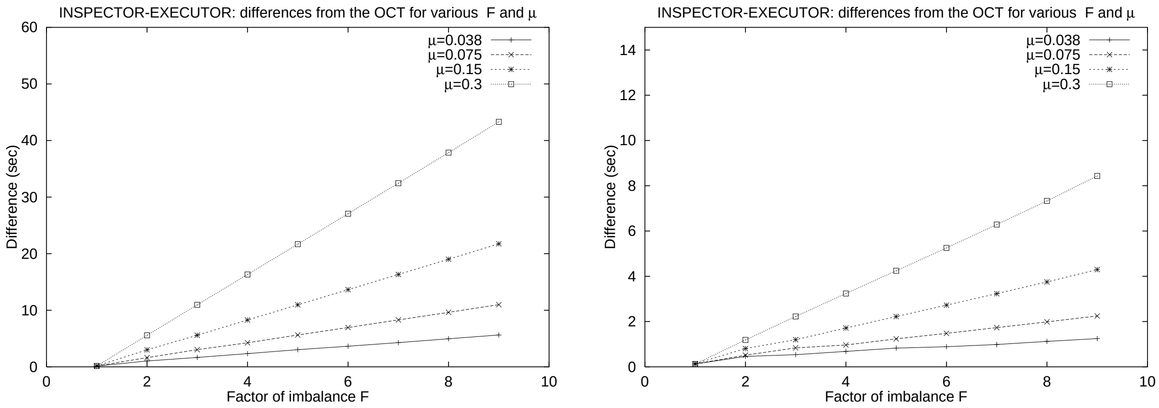


Figure 6: REDISTRIBUTION implementation: differences from the OCT for various values of μ as a function of the factor of imbalance F.



(a) (b)

Figure 7: INSPECTOR–EXECUTOR implementation: differences from the OCT for various values of μ as a function of the factor of imbalance F considering (a), or not (b), the first time step.

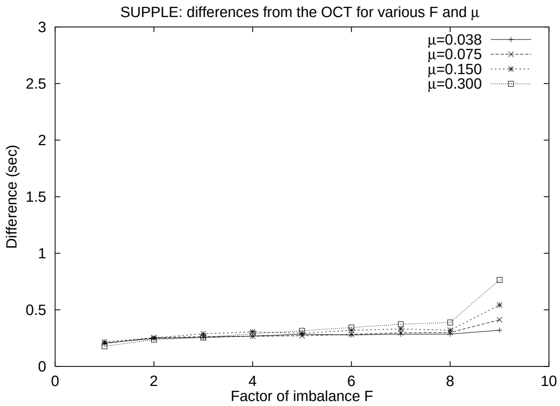


Figure 8: SUPPLE implementation: differences from the OCT for various values of μ as a function of the factor of imbalance F.

correspondingly receives a copy of the four adjacent partitions. A huge amount of data (28 MB) is thus moved at each time step to implement the first Convection loop. On the other hand, the slopes of the curves shown in Figure 5 indicate that the cyclic data layout does not perfectly balance processor workloads, as already shown in Figure 4.(b).

For $F = 9$, the differences from the OCT are equal to 0.37, 0.56, 0.92 and 1.63 *sec.* for μ equal to 0.038, 0.075, 0.15, 0.3 *msec.*, respectively. A residual imbalance thus affects each execution of the Reaction parallel loop. Moreover, since processors synchronize during the Convection phase, the imbalance is directly proportional to the number of time steps carried out (10 in the tests executed). Load imbalance might thus significantly decrease the efficiency of actual simulations, which require the execution of thousands of time steps.

Similar considerations can be made for the implementation (referred to as REDISTRIBUTION) which dynamically redistributes the simulation grid (block–to–cyclic and cyclic–to–block) at each time step before and after the execution of the Reaction computations. The differences from the OCT for this implementation are shown in Figure 6, and range from 0.47 to 0.64 seconds for $\mu = 0.038$ *msec.*, and from 0.47 to 1.89 seconds for $\mu = 0.3$ *msec.* In all the tests carried out, the completion times were about 0.27 seconds longer than the ones obtained with the CYCLIC implementation. A constant overhead is thus due to redistribute from block to cyclic the simulation grid at the end of the Convection phase, and to restore the block distribution at the end of the reaction phase. The total per–time–step communication volume is lower, in this implementation, than the CYCLIC one (about 16 MB instead of 28 MB), but the gains due to the lower communication volume and, especially, to the better data locality exploited by this solution, do not counterbalance the overheads that derive from the communications which implement redistribution. In the CYCLIC solution in fact, communications occur only among nearest–neighboring processors, while in the REDISTRIBUTION case all–to–all communications are needed which may cause network congestion.

The experimental results obtained running the INSPECTOR–EXECUTOR implementation are reported in Figure 7. Figure 7.(a) shows the differences from the OCT at the end of the tenth simulation time step, while the curves in Figure 7.(b) do not consider the times required to carry out the first time step. The *Inspector–Executor* implementation exploits, in fact, a block–distributed simulation grid and static iteration schedule to perform the first time step. During execution, information about variable iteration execution costs is collected. These costs are then used to build a *workload migration plan* which is exploited in the subsequent time steps to migrate excess work towards underloaded processors. In real flame simulations in which the number of time steps is very large, the greater cost of the first time step becomes negligible and thus the behavior of the *Inspector–Executor* implementation is better characterized by the curves reported in Figure 7.(b). However, even neglecting the first time step, the results were the worst obtained. The differences from the OCT range from 0.12 to 1.24 seconds for $\mu = 0.038$ *msec.*, and from 0.12 to 8.43 seconds for $\mu = 0.3$ *msec.* Two main factors jeopardize the effectiveness of the approach for the benchmark considered. Firstly, only iteration execution is profiled to derive processor workloads. The *workload migration plan* thus represents an ideal workload redistribution which does not consider the overheads incurred in actually performing the redistribution. Secondly, the *communication schedule* might not exactly respect the ideal *workload migration plan* because iterations have discrete costs while the *workload migration plan* redistributes the workload according to a continuous model.

Finally, the plot reported in Figure 8 shows the result obtained by exploiting the SUPPLE run–time support. One difference appears if we compare SUPPLE curves with the ones related to the other implementation strategies. While the other implementations resulted in execution times characterized by overheads which increase as the average iteration execution time μ and the factor of imbalance F are increased, SUPPLE behaves differently. The curves plotted in Figure 8 are nearly flat up to a factor of imbalance equal to 8, and show an increase in the overheads only for $F = 9$. The exploitation of the SUPPLE support allows us to balance the processors' workloads with a completion time which is very close to the OCT. The differences range from 0.20 to 0.32 seconds for $\mu = 0.038$ *msec.*, and from 0.18 to 0.76 seconds for $\mu = 0.3$ *msec.* With data sets characterized by a factor of imbalance equal to 8, the differences from the OCT are only 0.28, 0.30, 0.32 and 0.38 *sec.* for μ equal to 0.038, 0.075, 0.15, 0.3 *msec.*, respectively. These good results are mainly due to the fully distributed load balancing policy exploited by SUPPLE which adopts local policies of imbalance detection to avoid global synchronizations, and hides most dynamic scheduling overheads by overlapping communication with useful computations. The increase in the overheads measured for $F = 9$

Table III: Completion times (in seconds) of 10 simulation time steps for $\mu = 0.15$ and various F

Version	$F = 1$	$F = 2$	$F = 3$	$F = 4$	$F = 5$	$F = 6$	$F = 7$	$F = 8$	$F = 9$
CYCLIC	32.98	33.05	33.14	33.23	33.32	33.41	33.50	33.59	33.68
REDISTRIBUT.	33.25	33.31	33.40	33.49	33.58	33.67	33.76	33.85	33.94
INSPECT.–EXEC.	32.92	35.77	38.34	41.03	43.72	46.40	49.08	51.78	54.51
SUPPLE	32.98	33.02	33.05	33.07	33.06	33.09	33.10	33.09	33.31

must be due to *output contention* problems arising for truly highly unbalanced workloads that cause the network interface of the few overloaded processors to be congested by a message traffic that is too high.

6 Conclusions

We have discussed four different solutions for the implementation of a two–dimensional flame simulation code, a benchmark that has some interesting features that makes it representative of a large class of irregular applications:

- Constant stencil data references and uniform iteration costs in the Convection phase of the simulation. These features encourage static binding of data and computations, with specific attention to locality exploitation.
- Non–uniform and unpredictable computation costs in the Reaction phase, with a large part of the total load concentrated in a few regions of the simulation grid. This encourages the use of scattered data distributions which jeopardize locality exploitation, or of dynamic/hybrid scheduling strategies which allow the binding of data and computations to the processors involved to occur only at run–time, on the basis of the current workload distribution.

The first solution we tested (CYCLIC) is completely static, and adopts a cyclic data layout to achieve the goal of load balance. The second implementation (REDISTRIBUTION) uses a block distribution during the first loop, but the arrays are redistributed in a cyclic layout before executing the unbalanced Reaction computations. The third solution is based on the INSPECTOR–EXECUTOR paradigm and can be classified as a hybrid since the actual binding of data and computation may be changed at run–time after a synchronization carried out at fixed points. Finally, the fourth approach exploits the SUPPLE run–time support which adopts a statically fixed block distribution, and asynchronously migrates chunks of iterations and associated data at run–time to balance the load.

All four solutions were implemented on the top of the same MPI message passing layer, while, where possible, the same optimizations were applied. The experiments were conducted on a Cray T3E, a high–performance distributed–memory architecture with a very efficient communication and synchronization subsystem. We employed different synthetic data sets built on the basis of a simple load imbalance model. These data sets, fed in input to the flame simulation code, introduce different known workload imbalances.

The best results were obtained with the SUPPLE implementation which tries to exploit locality by starting scheduling iterations according to a statically chosen scheduling policy, and asynchronously migrates iterations only if an actual load imbalance is detected. Although quite good, the results obtained with the static implementation based on a cyclic data distribution are worse than the ones with SUPPLE. Even worse results were obtained with the REDISTRIBUTION and INSPECTOR-EXECUTOR implementations. A summary of the completion times obtained with all the implementations for $\mu = 0.15$ and various F is reported in Table III.

Finally, we have assessed the worse results obtained by the INSPECTOR-EXECUTOR implementation with respect to SUPPLE. The former solution tries to reduce overheads by minimizing the number of messages, but it does not allow overlapping the communication time with useful computations. Moreover, it entails introducing synchronization points between the execution of consecutive parallel loops, with all-to-all communications to create a global knowledge about the workload distribution.

On the other hand, SUPPLE uses many small messages, does not introduce synchronization points, and allows many techniques to be exploited for overlapping useful computations with communication latencies. However, since SUPPLE employs many small messages, overheads due to message sending, receiving and probing need to be very small, as on our target architecture, the Cray T3E. Future work should verify whether the results above either remain the same or change dramatically when different architectures, where communication overheads are much larger, are adopted.

We believe that hybrid strategies like the one adopted by SUPPLE can be profitably exploited in many cases where locality exploitation and load imbalance are contrasting goals, and the workload distribution is unpredictable. The compilation model of data–parallel languages should be extended to exploit SUPPLE–like run–time supports, which handle more efficiently loops with non–uniform and unpredictable computation times.

Moreover, this extension should also be valuable for when the load imbalance derives from the system, as happens in multiprogrammed or heterogeneous parallel systems. The use of supports which decide the binding of computations and data at run–time, should allow the compiler to produce more efficient codes for these machines, even for uniform computations. In fact, completely static solutions are not practicable when load imbalance cannot be predicted, i.e. when a mixture of interactive and batch parallel jobs are processed by a multiprogrammed machine.

Acknowledgments

We acknowledge the support of the CINECA Consortium of Bologna for the grant which allowed us to make much use of their computational facilities, in particular the Cray T3D and T3E multiprocessors.

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] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *J. of Parallel and Distr. Comp.*, 22(3):462–479, 1994.

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

[4] High Performance Fortran Forum. *HPF-2 Scope of Activities and Motivating Applications*, Nov. 1994. Version 0.8.

[5] High Performance Fortran Forum. *High Performance Fortran Language Specification*, Jan. 1997. Version 2.0.

[6] 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.

[7] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluating Compiler Optimizations for Fortran D. *J. of Parallel and Distr. Comp.*, 21(1):27–45, April 1994.

[8] 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.

[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] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of block algorithms. In *4th Int. Conf. on Architectural Support for Progr. Lang. and Operating Systems*, pages 63–74, Santa Clara, CA, April 1991.

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

[12] 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.

[13] S. Orlando and R. Perego. A template for non–uniform parallel loops based on dynamic scheduling and prefetching techniques. In *Proc. of the 1996 ACM Int. Conf. on Supercomputing*, pages 117–124, 1996.

[14] S. Orlando and R. Perego. SUPPLE: an Efficient Run–Time Support for Non–Uniform Parallel Loops. Technical Report TR-17/96, Dipartimento di Mat. Appl. ed Informatica, Università di Venezia, Dec. 1996.

[15] S. Orlando and R. Perego. A Support for Non–Uniform Parallel Loops and its Application to a Flame Simulation Code. In *Proc. of IRREGULAR '97*. LNCS Spinger-Verlag, 1997. To appear.

[16] G. Patnaik, K. J. Laskey, K. Kailasanath, E. S. Oran, and T. A. Brun. Flic – a detailed two–dimensional flame model. Memorandum Report 6555, Naval Research Laboratory, Sept. 1989.

[17] O. Plata and F. F. Rivera. Combining static and dynamic scheduling on distributed–memory multiprocessors. In *Proceedings of the 1994 ACM Int. Conf. on Supercomputing*, pages 186–195, 1994.

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

[19] R. Ponnusamy, J. Saltz, A. Choudary, Y-S Hwang, and G. Fox. Runtime Support and Compilation Methods for User-Specified Irregular Data Distributions. *IEEE Trans. on Parallel and Distr. Systems*, 6(8):815–831, Aug. 1995.

[20] D. A. Reed, L. M. Adams, and M. L. Patrick. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *IEEE Transactions on Computers*, 36(7):845–858, July 1987.

[21] J. Saltz et al. Runtime and Language Support for Compiling Adaptive Irregular Programs on Distributed Memory Machines. *Software Practice and Experience*, 25(6):597–621, June 1995.

[22] J. Saltz et al. Runtime Support and Dynamic Load Balancing Strategies for Structured Adaptive Applications. In *Proc. of the 1995 SIAM Conf on Par. Proc. for Scientific Computing*, Feb. 1995.

[23] J.M. Sipelstein and E. Blelloch. Collection-Oriented Languages. *Proc. of the IEEE*, 79(4):504–523, April 1991.

[24] 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.

[25] 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.

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