Revised paper submitted to the *Journal of Parallel and Distributed Computing*, Feb 98

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

Salvatore Orlando*, Raffaele Perego†

* *Dip. di Matematica Applicata ed Informatica*
Università Ca' Foscari di Venezia
Venezia Mestre - Italy
orlando@unive.it

† Istituto CNUCE
Consiglio Nazionale delle Ricerche (CNR)
Pisa, Italy
r.perego@cnuce.cnr.it

CORRESPONDING AUTHOR:

Raffaele Perego, Istituto CNUCE, Consiglio Nazionale delle Ricerche (CNR), via S. Maria 36, Pisa, 56126 Italy. Tel. +39 50 593253, Fax +39 50 904052, E-mail: r.perego@cnuce.cnr.it

RUNNING HEAD: Non–Uniform Data–Parallel Computations

Abstract

Data–parallel languages allow programmers to easily express parallel computations by means of high-level constructs. 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 adaptive 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) scheduling 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.

LIST OF SYMBOLS

We use `typewriter` font, generated using the LaTeX `\tt` command, to denote language constructs and subroutine names. All the symbols in formulas are generated using the standard LaTeXmathematical environment.

1 INTRODUCTION

High–level data–parallel languages make it easier to write parallel programs for both shared and distributed–memory multiprocessors. Exploiting the results of previous research projects [5, 24], a consortium of academic and industrial partners (participating in the *High Performance Fortran Forum*) defined the HPF language [3]. HPF allows programmers to provide a few directives to specify processor and data layouts, and to express parallelism by means of parallel loop constructs or collective operations. The compiler manages data distribution and 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.

In this paper we address the load balancing problem of *non–uniform* data–parallel computations which exploit regular and predictable data references. These computations are usually expressed by means of parallel loops 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 both data to memories and computations to processors 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 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.

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 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 [17, 22, 7]. Also on UMA multiprocessors, however, dynamic iteration assignment should be guided not only by considering the load balance goal, but also data reuse and locality exploitation [11].
Dynamic approaches can also be adopted on distributed–memory, message passing machines. 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 implementations of the dispatcher have been proposed to reduce overheads and memory requirements [10].

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 on the basis of the costs measured for previous iterations, or asynchronously on the basis of the decisions taken by the dynamic scheduling policy used. In the former case, possible variations in the workload distribution may affect the efficiency of the method, and communication latencies are harder to hide because computation restarts only after the synchronization phase that redistributes processor workloads. 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 the computations statically assigned to it, and a worker, which self–schedules computations belonging to either its local queue or queues managed by other processors.

In the rest of the paper we will compare alternative implementation strategies which can be used to balance the workloads derived from the execution of non–uniform parallel loops. Due to the prohibitive memory requirements and the high overheads involved, we will not consider completely dynamic approaches and we will concentrate our attention on static and hybrid scheduling policies only. The application we consider as a benchmark is a two–dimensional flame simulation code, an irregular code where two parallel loops that access the same arrays must be executed several times to carry out consecutive steps of the simulation. 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. 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 experimented 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.

```
C  Loop on the time steps
   DO k= 1,K
C     Convection phase over a structured mesh
      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
      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 flame modeling code.

2 THE BENCHMARK

The benchmark we used to evaluate the various approaches, is derived from a class of scientific codes used to carry out detailed flame simulations [15]. A code that performs a time-dependent multi-dimensional simulation of hydrocarbon flames is described in [18], and is also included in the HPF–2 draft [4] as one of its motivating applications. The 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. Fig. 1 shows the HPF-like code of the kernel of the simplified two-dimensional flame modeling program used as benchmark. 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 [4].

The most interesting characteristic of this code is that, in order to efficiently exploit a distributed–memory 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 exploit a different distribution or dynamic scheduling techniques in order to balance the highly variable workloads during the Reaction phase.

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.

CYCLIC. 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.

REDISTRIBUTION. In the second static approach the block layout used during the first loop 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 redistribution occurs at given instants during execution.

I/E. The first hybrid approach is based on the Inspector-Executor paradigm [2]. While a block distribution is used during the first Convection loop, a partial redistribution takes place at every time step before entering the Reaction phase. However, the actual binding of redistributed data and computations is decided dynamically, on the basis of load information collected at run-time.

SUPPLE. The second hybrid approach exploits the SUPPLE support (SUPport for Parallel Loop Execution) [12, 14]. It adopts a statically fixed block distribution, but, during execution, chunks of iterations and associated data may be dynamically migrated toward underloaded processors. In this case both the time instants when a possible migration may occur, and the actual migration plan, are decided 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.

3.1 Implementation of the convection phase

The Convection loop is uniform and accesses data with a regular and constant *Five-Point stencil* (see Fig. 1). The first solution discussed in this paper uses a cyclic data layout for this phase, while the others exploit a block data layout to reduce the number of accesses to remote data needed to implement stencil references.

Block data layout provides that 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. 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 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 [6, 19]. 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. Moreover, to overlap communications with useful computations, iterations are reordered: the iterations that assign data items belonging to the inner area and that refer local data only, are scheduled between the asynchronous sending of the perimeter area to neighboring processors and the receiving of the corresponding data into the ghost area.

The execution of the iterations that assign the elements lying on the inner area is also optimized. We group these iterations into *chunks* of a fixed size. To this end, each inner block partition is statically *tiled*. Tiling is a useful optimization to improve locality [9], and is also exploited by both the hybrid implementations of the non–uniform Reaction loop. In fact to reduce load balancing overheads, chunks and associated data tiles instead of single iterations are migrated on the basis of the workload distribution. Finally, since the results of remote chunk 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 is read by the first Convection loop. To maintain data coherence, a *full/empty* flag [1] is associated with each data tile. If a chunk that modifies a given tile is migrated, then the associated flag is set. The flag is unset when the updated data tile is received back from the remote processor. This solution avoids the introduction of a barrier synchronization at the end of the Reaction loop to wait for coherence to be restored. By checking *full/empty* flags only when a tile has to be read, we ensure data coherence and allow 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 computations with communications. In fact, a generic processor p that executes a loop iteration has to read a five point stencil of array `B`, but all the points except the center of the stencil are owned by other processors. In particular, because in all our experiments we exploited a logical two–dimensional processor grid, all remote points are owned by the four neighbors 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 [6] can be applied to minimize communication overheads. As a consequence of these optimizations, at each time step, before executing the Convection loop each processor $p_{h,k}$, with position (h,k) in the two–dimensional logical grid, sends its partition of the array `B` to its four neighbors $p_{h-1,k}$, $p_{h+1,k}$, $p_{h,k-1}$, $p_{h,k+1}$, and correspondingly receives from them a copy of their partitions of `B`. At this point, $p_{h,k}$ can carry out the Convection loop 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 $\mathtt{B}_{h-1,k}$, 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. The simplest way to achieve a good workload balance is to give up the idea of exploiting data locality in the Convection loop discussed above 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.

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 loop, 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 in this case requires *all-to-all* communications. To reduce the number of messages, data for transmission to the same processor are aggregated, 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 *contention* problems which might arise if several processors simultaneously send messages to the same destination.

I/E. The solution to many problems whose irregularities prevent pure compile-time optimizations can be addressed by exploiting *Inspector-Executor* (I/E) codes which collect information at run-time, and then use it to optimize subsequent computations [2, 20]. If we refer to the code in Fig. 1, we can outline how an I/E approach can be exploited in this case [21, 18].

1. During each iteration k of the outer loop, processors profile the Reaction loop execution and store iteration costs. At the end of the loop each processor p_i knows the costs of its own loop iterations and its total local load $TL(p_i)$. $TL(p_i)$ is then broadcast to all the other processors.

2. Once received the locally computed loads from all the other processors, each processor autonomously decides if balancing actions have to be taken. A simple heuristic method which measures the difference between the loads of the most heavily and lightly loaded processors, and compares it with a fixed *threshold* is used for this purpose. If a load imbalance is detected, all the processors independently execute a *binpack* algorithm [21] to build the same suboptimal *workload migration plan*, which establishes how much work should be moved from each overloaded processor to each underloaded one to achieve load balance. Suppose that $\overline{M}$ is the average processor load, and that 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 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 each p'_i, $\sum_{j=0}^{n-1} \alpha^i_j \approx TL(p'_i) - \overline{M}$, and, for each $\overline{p}_j$, $\sum_{i=0}^{m-1} \alpha^i_j \approx \overline{M} - TL(\overline{p}_j)$. This is achieved by choosing a pair of processors $(p'_i, \overline{p}_j)$, and by computing the amount of load, i.e. $max(TL(p'_i) - \overline{M}, \overline{M} - TL(\overline{p}_j))$, that should be migrated from p'_i to $\overline{p}_j$. If p'_i's workload is

greater than $\overline{M}$ also after this workload migration, another underloaded processor $\overline{p}_j$ is chosen, and the process is repeated. Conversely, if p'_i's workload is not sufficient to bring the load of $\overline{p}_j$ up to $\overline{M}$, the process is repeated by choosing another overloaded processor p'_i. To reduce the number of workload redistributions, a heuristic policy which chooses the most overloaded and most underloaded processors first is used to select the processor pairs involved in each workload migration.

3. On the basis of the *workload migration plan* built at the previous step, overloaded processors choose n bunches of iterations $\beta^i_0, \beta^i_1, \cdots, \beta^i_{n-1}$ to be migrated. The costs of these bunches must be approximatively $\alpha^i_0, \alpha^i_1, \cdots, \alpha^i_{n-1}$. To reduce the communication volume, the dimensions of the bunches have to be minimized. To this end, each p'_i sorts its iteration indexes on the basis of the costs previously measured, and chooses heavier iterations first to fill the bunches β^i_j, $j = 0, 1, \cdots, n-1$. The resulting n couples $(\overline{p}_j, \beta^i_j)$ constitutes the *communication schedule* of overloaded processors 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 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 waits for 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 may vary as simulation progresses, processors must continue profiling the non–uniform parallel loop during all the simulation and recompute, if it is needed, a new *workload migration plan* and *communication schedule*. Each outer loop iteration thus requires a collective communication to broadcast the current processor loads. This global knowledge of processor loads, although it allows to minimize data and computation reallocation, is expensive to maintain. In addition, the cost of building both the *workload migration plan* and *communication schedule* is not negligible. To reduce this cost an $O(n)$ algorithm for the approximate sorting of iteration costs was proposed [21] (see step 3 of the above implementation scheme). However, we followed a different direction which consists in treating a chunk of iterations, rather than a single iteration, as the scheduling unit. This reduces 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 I/E 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.

SUPPLE. SUPPLE is a run-time support for the implementation of both uniform and non–uniform parallel loops [12, 14]. It only supports block-distributed arrays to exploit the locality that derives 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 a 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 [16], 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 iteration chunks stored in a *local* queue Q. Once a processor understands that its *local* queue Q is *becoming empty*, the dynamic part of the scheduling policy begins. Exploiting a *receiver initiated* load balancing technique [8, 23], SUPPLE tries to balance the processor workloads by migrating chunks and corresponding data tiles from overloaded to underloaded processors. Migrated chunks and data tiles are stored by each receiving processor in a *remote* queue RQ. Dynamic scheduling decisions are taken on the basis of local information only as follows:

- During the initial static scheduling phase, each processor executes local chunks stored in Q and measures their execution time to estimate the average cost of its own chunks. Moreover, when chunks are migrated toward underloaded processors, their average cost is communicated as well. On the basis of this knowledge, processors can thus estimate their own *current load* at any time by inspecting their queues Q and RQ.

- When the estimated *current 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. Correspondingly, the processors which receive a chunk migration request will grant the request only if their *current load* is higher than the same *Threshold*. Note that the *Threshold* parameter must be 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 local *round-robin* policy. In fact, it is often more important not to spend too much time in making a decision than always making the best decision [8]. However, to reduce the overheads which might derive from migration requests which cannot be served, 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 to select source processors skips those processors that have already 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 slightly modified *Factoring* [7] scheme to locally determine this number: 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.

The policy exploited by SUPPLE to manage data coherence and termination detection is fully distributed and asynchronous. The *full/empty* flag technique 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 (in our case during the next execution of the Convection loop), 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 the Reaction loop execution: a new loop, in our case the Convection loop of a new time step, 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 Section, we built a set of synthetic input data sets for the flame simulation benchmark whose skeleton is shown in Fig. 1. The synthetic data sets are 1024×1024 arrays. They represent possible states of the simulation grid characterized by different workload distributions: the per–point time needed to perform the Reaction phase (function `Reaction`) has been in fact forced to be directly proportional to the value of the corresponding data set point.

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 $t \geq d$. The *high*–cost points were positioned following an exponential distribution with respect to two points of the simulation grid. Fig. 2 shows a representation of a data set characterized by $d = 0.1$ in which *high*–cost (*low*–cost) points are displayed as black (white). 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 $F = \frac{t}{d}$. 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 balanced data set since the computational cost associated with each grid point is exactly equal to μ.

It can be shown that F characterizes the associated data set by furnishing a upper bound to the possible *slowdown factor* deriving from its unbalanced workload distribution when the data set is equally and statically distributed among the processors. In fact, if S is the size of the data set and P is the number of processors employed, we have $\mu = \frac{T}{S}$, while the optimal completion time is equal to $\frac{T}{P} = \mu \cdot \frac{S}{P}$. Considering that the *high*–cost points are $d \cdot S$, their cost is thus $\overline{\mu} = \frac{t \cdot T}{d \cdot S} = F \cdot \frac{T}{S} = F \cdot \mu$. In the *worst case* all the points assigned to a given processor $\overline{p}$ have high cost. If this is the case and the data set is equally distributed among the processors, the execution time of $\overline{p}$ in absence of load balancing is $\overline{\mu} \cdot \frac{S}{P} = F \cdot \mu \cdot \frac{S}{P} = F \cdot \frac{T}{P}$. This time, which clearly dominates the overall program completion time, is exactly F times the optimal completion time.
Note that our characterization of the workload imbalance is independent of P. Other measures of load imbalance, such as the difference between the workloads of the most loaded and the most underloaded processors, depends not only on the specific data set, but also on the number of processors exploited.

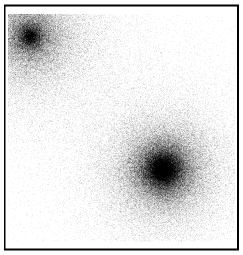


Figure 2: Representation of a synthetic data set.

Table 1: Per–point reaction cost time for some values of μ and F

μ	F	*high*–cost pts	*low*–cost pts
0.038 $msecs$	2	0.076 $msecs$	0.033 $msecs$
0.15 $msecs$	2	0.3 $msecs$	0.133 $msecs$
0.038 $msecs$	5	0.19 $msecs$	0.021 $msecs$
0.15 $msecs$	5	0.75 $msecs$	0.083 $msecs$

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 various versions of the benchmark so that a partition of 128×128 points is assigned to each processor. The points of each partition 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.

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

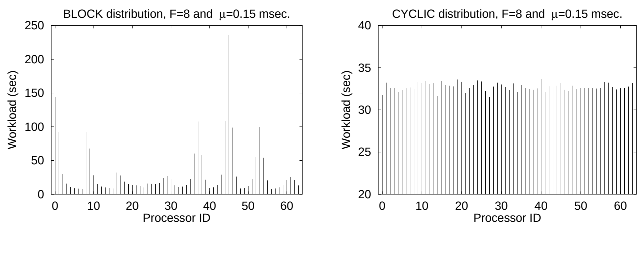


Figure 3: Per-processor workload for *block* (a) and *cyclic* (b) distributions of the 1024×1024 simulation grid on the 8×8 processor grid: case $F = 8$ and $\mu = 0.15$ msecs.

is thus $\frac{\mu}{3}$. We conducted experiments with input data sets characterized by several values of μ and, specifically, for $\mu = 0.038$ *msecs*, 0.075 *msecs*, 0.15 *msecs* and , 0.3 *msecs*. If we fix the number of simulation steps executed (10 in our experiments), we can estimate the Optimal Completion Time (OCT) of the whole simulation (OCT equal to 8.19, 16.38, 32.76, 65.53 *secs*, for μ equal to 0.038, 0.075, 0.15, 0.30 *msecs*, respectively). This time, which does not take into account any overheads, can be 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).

Table 1 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.

Fig. 3 shows the per–processor workloads that derive from *block* (Fig. 3.(a)) and *cyclic* (Fig. 3.(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 $\mu = 0.15$ *msecs*. 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. On the other hand the workload distribution deriving from the *cyclic* data layout is suboptimal. Processor workloads range, in this case, from 31.52 to 33.59 *secs* (OCT = 32.76 *secs*).

Figures 4 and 5 report the results of the experiments conducted to evaluate the four implementation strategies discussed. Each plot regards a different implementation and shows several curves related to distinct values of μ. Each curve plots the difference between the completion time of the slowest processor and the theoretical OCT as a function of F. This difference can be considered as the sum of all overheads due to communications, residual workload imbalance, and loop housekeeping. It also includes the time to send/receive messages that implement the stencil communications of the first convection loop. 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 Fig. 4.(a) range from 0.20 to 0.37 seconds for $\mu = 0.038$ *msecs*, and from 0.21 to 1.63 seconds for $\mu = 0.3$ *msecs*. On the one hand, these results show that the optimization techniques adopted 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 correspondingly receives a copy of the four adjacent partitions. A huge amount of data (28 MB) is thus moved at each time step. On the other hand, the slopes of the curves shown in Fig. 4.(a) indicate that the cyclic data layout does not perfectly balance processor workloads, as already shown in Fig. 3.(b). For $F = 9$, the differences from the OCT are equal to 0.37, 0.56, 0.92 and 1.63 *secs* for μ equal to 0.038, 0.075, 0.15, 0.3 *msecs*, respectively. A residual imbalance thus still affects each execution of the Reaction parallel loop.

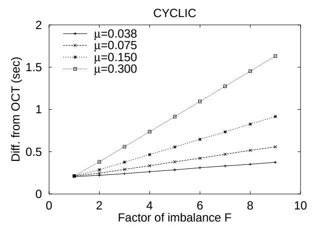
Similar considerations can be made for the REDISTRIBUTION implementation which dynamically redistributes the simulation grid at each time step before and after the execution of the Reaction computations. The differences from the OCT for this implementation are shown in Fig. 4.(b), and range from 0.47 to 0.64 seconds for $\mu = 0.038$ *msecs*, and from 0.47 to 1.89 seconds for $\mu = 0.3$ *msecs*. In all the tests carried out, the completion times were larger than the CYCLIC ones: a constant overhead is 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 in 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 more expensive *all-to-all* communications are needed.

The experimental results obtained running the I/E implementation are reported in Fig. 5.(a). These results are the worst we obtained. This is partially due to the small number of simulation steps executed in the tests. The I/E implementation in fact is not able to balance the load during the first simulation step which is used to collect load information. 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 I/E implementation should be better. However, even neglecting the first time step, the differences from the OCT range from 0.12 to 1.24 seconds for $\mu = 0.038$ *msecs*, and from 0.12 to 8.43 seconds for $\mu = 0.3$ *msecs*. 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.

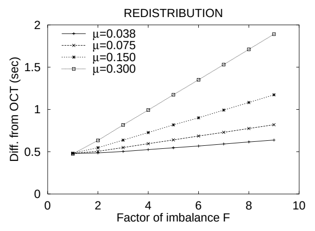
Finally, the plot reported in Fig. 5.(b) shows the result obtained by exploiting SUPPLE. One difference appears if we compare these curves with the others. While the other implementations resulted in execution times characterized by overheads which increase proportionally to μ and F, the curves plotted in Fig. 5.(b) 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 $\mu = 0.038$ *msecs*, and from 0.18 to 0.76 seconds for $\mu = 0.3$ *msecs*. 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 *secs* for μ equal to 0.038, 0.075, 0.15, 0.3 *msecs*, respectively. The increase in the overheads measured for $F = 9$ is due to unavoidable *output contention* problems arising for truly highly unbalanced workloads that cause the network interface of the few overloaded processors to be congested by too many migration requests.

6 CONCLUSIONS

We have discussed four different solutions for the implementation of a two–dimensional flame simulation code. This code was also included in the HPF-2 draft [4] as one of its motivating applications because its features, as it is claimed in the draft, make the application hardly supported by HPF-1–compliant compilers, so that sophisticated *adaptive* techniques should be required to deal with them.

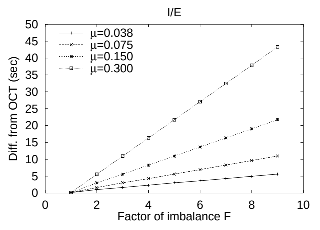


(a)

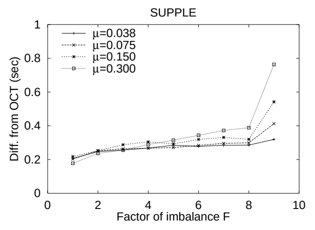


(b)

Figure 4: Differences from the OCT for various values of μ as a function of F for the CYCLIC (a), and REDIST. (b) implementations.



(a)



(b)

Figure 5: Differences from the OCT for various values of μ as a function of F for the I/E (a), and SUPPLE (b) implementations.

The first solution we tested is completely static, and adopts a cyclic data layout to balance the workloads. The second implementation uses a block distribution during the first loop, but the arrays are redistributed according to a cyclic layout before executing the unbalanced chemical reaction computation.
The third solution is based on the I/E paradigm, and can be classified as a hybrid one since the initial static binding of data and computations can be modified at run-time to balance the workloads and obtain better performances. More specifically, in this case the actual migration of part of data and computations is dynamically decided after a synchronization carried out at fixed points to collect global workload information.
Finally, the fourth approach is also hybrid, and exploits our SUPPLE run–time support. SUPPLE exploits locality by initially scheduling iterations according to a static scheduling policy and a block data layout, and asynchronously

Table 2: Completion times (in seconds) of 10 simulation time steps for $\mu = 0.15$ and various F

Version	$F = 1$	$F = 2$	$F = 4$	$F = 6$	$F = 8$
CYCLIC	32.98	33.05	33.23	33.41	33.59
REDIST.	33.25	33.31	33.49	33.67	33.85
I/E	32.92	35.77	41.03	46.40	51.78
SUPPLE	32.98	33.02	33.07	33.09	33.09

migrates iterations only if an actual load imbalance is detected.

All the 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. 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 of the benchmark. Quite good results were also obtained by the static implementation, which simply exploits a cyclic data layout, thus proving that the use of static compiler optimizations results to be effective even for an application that has been inserted in the HPF2 draft to motivate the need of more complex and adaptive run-time supports like CHAOS [20]. Worse results were instead achieved by the hybrid I/E implementation and by the static implementation which exploit array redistribution. A summary of the completion times obtained with all the implementations for $\mu = 0.15\ msecs$ and various F is reported in Table 2.

Particularly interesting are the results obtained by the two adaptive approaches, I/E and SUPPLE. The I/E solution tries to reduce overheads by minimizing number and volume of messages required to balance the workloads, but it does not allow communications to be overlapped with useful computations, and introduces synchronization points between the execution of consecutive parallel loops. SUPPLE instead uses smaller messages to migrate work toward underloaded processors because only local knowledge is exploited to make fast load balancing decisions, and some of the decisions taken might result to be wrong if larger amounts of work were migrated. On the other hand, SUPPLE does not introduce synchronization points, is fully distributed and asynchronous, and allows many techniques for hiding communication latencies to be exploited. Moreover SUPPLE can be used for the implementation of non–uniform parallel loops executed only once (non–uniform single iterated parallel loops are common in many high–level vision and computer graphics applications), or iterated loops where the workload distribution is highly varying. Conversely, in order to apply the I/E paradigm, the same non–uniform parallel loop must be executed many times and, more importantly, the load distribution has to vary very slowly between successive loop executions to effectively reuse the historical load information.

In conclusion, hybrid scheduling techniques like the one adopted by SUPPLE can be profitably exploited in many cases where locality exploitation and load imbalance are two contrasting goals, and the workload distribution is unpredictable. Moreover, their use is valuable even for uniform computations executed on multiprogrammed or heterogeneous parallel systems where the load imbalance is caused by variations in capacities of processing nodes [13]. We believe that in all these cases the compilation model of data–parallel languages should be extended to exploit SUPPLE–like techniques, which may modify the binding of part of computations and data at run–time, thus handling unpredictable load imbalances.

References

[1] R. Alverson et al. The Tera computer system. In *Proc. of the 1990 ACM ICS*, 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. *JPDC*, 22(3):462–479, 1994.

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

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

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

[6] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluating Compiler Optimizations for Fortran D. *JPDC*, 21(1):27–45, 1994.

[7] S.F. Hummel, E. Schonberg, and L.E. Flynn. Factoring: A Method for Scheduling Parallel Loops. *CACM*, 35(8):90–101, 1992.

[8] V. Kumar, A.Y. Grama, and N. Rao Vempaty. Scalable Load Balancing Techniques for Parallel Computers. *JPDC*, 22:60–79, 1994.

[9] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of block algorithms. In *Proc. of the 4th ACM ASPLOS*, pages 63–74, Santa Clara, CA, Apr. 1991.

[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 TPDS*, 5(4):379–400, Apr. 1994.

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

[13] S. Orlando and R. Perego. Scheduling Data–Parallel Computations on Heterogeneous and Time–Shared Environments. Technical Report TR-16/97, Dip. di Mat. Appl. ed Informatica, Università di Venezia, Sept. 1997.

[14] S. Orlando and R. Perego. A Support for Non–Uniform Parallel Loops and its Application to a Flame Simulation Code. In *Proc. of the 4th Int. Symposium, IRREGULAR '97*, pages 186–197, Paderborn, Germany, June 1997. LNCS 1253, Spinger-Verlag.

[15] 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 Lab., Sept. 1989.

[16] O. Plata and F. F. Rivera. Combining static and dynamic scheduling on distributed–memory multiprocessors. In *Proc. of the 1994 ACM ICS*, pages 186–195, 1994.

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

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

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

[20] 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.

[21] 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.

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

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

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

Biographies

SALVATORE ORLANDO received a Laurea degree cum laude and a Ph.D. degree in Computer Science from the University of Pisa in 1985 and 1991, respectively. He is currently an assistant professor at the University of Venice. His research interests include parallel languages, parallel computational models, optimizing and parallelizing tools, parallel algorithm design and implementation.

RAFFAELE PEREGO received his Laurea degree in Computer Science from the University of Pisa in 1985. He has been a contract professor of computer science at the University of Pisa since 1989. He is currently researcher at the Italian National Research Council. His research interests include scheduling, parallel algorithm design, parallel languages and tools, high performance computing.