# A Support for Non–Uniform Parallel Loops and its Application to a Flame Simulation Code

Salvatore Orlando[1] and Raffaele Perego[2]

[1] Dipartimento di Mat. Appl. ed Informatica - Università Ca' Foscari di Venezia, Venezia Mestre, 30173 Italy - email: orlando@unive.it.
[2] CNUCE, Consiglio Nazionale delle Ricerche (CNR), Pisa, 56126 Italy - email: r.perego@cnuce.cnr.it.

**Abstract.** This paper presents SUPPLE (SUPport for Parallel Loop Execution), an innovative run–time support for parallel loops with regular stencil data references and non–uniform iteration costs. SUPPLE relies upon a static block data distribution to exploit locality, and combines static and dynamic policies for scheduling non–uniform iterations. It adopts, as far as possible, a static scheduling policy derived from the owner computes rule, and moves data and iterations among processors only if a load imbalance actually occurs. SUPPLE always tries to overlap communications with useful computations by reordering loop iterations and prefetching remote ones in the case of workload imbalance. The SUPPLE approach has been validated by many experimental results obtained by running a multi-dimensional flame simulation kernel on a 64–node Cray T3D. We have fed the benchmark code with several synthetic input data sets built on the basis of a load imbalance model, and we have compared our results with those obtained with a CRAFT Fortran implementation of the benchmark.

## 1 Introduction

Data parallelism is the most common form of parallelism exploited to speed-up scientific applications. In the last few years, high level data parallel languages such as High Performance Fortran (HPF) [2], Vienna Fortran [16] and Fortran D [4] have received considerable interest because they facilitate the expression of data parallel computations by means of a simple programming abstraction. In particular HPF, whose definition is the fruit of several research proposals, has been recommended as a standard by a large Forum of universities and industries. Using HPF, programmers only have to provide a few directives to specify processor and data layouts, and the compiler translates the source program into an SPMD code with explicit interprocessor communications and synchronizations for the target multicomputer. Data parallelism can be expressed above all by using collective operations on arrays, e.g. collective Fortran 90 operators, or by means of parallel loop constructs, i.e. loops in which iterations are declared as independent and can be executed in parallel.

The typical HPF run-time support for parallel loops exploits a static data layout of arrays onto the network of processing nodes, and a static scheduling of iterations which depends on the specific data layout. The adoption of a static policy to map data and computations reduces run-time overheads, because all mapping and scheduling decisions are taken at compile–time. While it produces very efficient implementations for regular concurrent problems, the code produced for *irregular problems* (i.e. problems where some features like non–local data references and iteration costs cannot be predicted until run-time) may be characterized by poor performance. Many researches have been conducted in the field of run-time supports and compilation methods to efficiently implement irregular concurrent problems, and these researches are at the basis of the new proposal of the HPF Forum for HPF2 [3].

In this paper we address particular irregular problems that can be programmed by means of non–uniform parallel loops, i.e. loops where the iteration cost varies considerably and cannot be predicted at compile time. We present a truly innovative support called SUPPLE (SUPport for Parallel Loop Execution) to implement these loops. SUPPLE is not a general support on which we can compile every HPF parallel loop, but only non–uniform (as well as uniform) parallel loops with regular stencil data references. Since stencil references are regular and known at compile time, optimizations such as *message vectorization*, *coalescing* and *aggregation*, as well as *iteration reordering* can be carried out to reduce overheads and hide communication latencies [5]. SUPPLE is able to initiate loop computation using a statically chosen BLOCK distribution, thus starting iteration scheduling according to the *owner computes rule*. During the execution, however, if a load imbalance actually occurs, SUPPLE adopts a dynamic scheduling strategy that migrates iterations and data from overloaded to underloaded processors in order to increase processor utilization and improve performances. SUPPLE overlaps most overheads deriving from dynamic scheduling, i.e. messages to monitor loads and move data, with useful computations. The main features that allow us to overlap the otherwise large overheads are (1) *prefetching* of remote loop iterations on underloaded processors, and (2) a complete asynchronous and distributed support that does not introduce barriers between consecutive iterations of loops.

SUPPLE has been validated through the implementation of a multi-dimensional flame simulation code [13], which can be programmed by means of parallel loops characterized by regular stencil data reference and non–uniform iteration costs. One example of code that performs a time-dependent multi-dimensional simulation of hydrocarbon flames is described in detail in [12]. It is also included in the HPF–2 draft [3] as one of its motivating application codes. This code is split into two distinct computational phases executed at each time step. The first phase computes fluid *Convection* over a structured mesh, and is expressed by a uniform parallel loop with regular stencil data references. 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 cost may vary significantly since the chemical combustion proceeds at different rates across the simulation space. It can be thus implemented with a non–uniform parallel loop that refers local data only. 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 workload distribution evolves as the simulation progresses [12]. The HPF-like code of a simplified two-dimensional flame modeling code is reported in Fig. 1. This is the kernel code we used to validate SUPPLE.

```
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
```

**Fig. 1.** HPF-like code of a simplified two-dimensional flame modeling code.

The paper is organized as follows. Section 2 gives an overview of our support and presents the SUPPLE implementation of the flame simulation kernel. Assessments of the experiment results are reported and discussed in Section 3, while Section 4 draws the conclusions.

## 2 SUPPLE and its implementation of the flame simulation code

SUPPLE is a run-time support for the efficient implementation of data parallel loops with regular, *stencil* references, characterized by per-iteration computation times which may be non–uniform, thus generating a workload imbalance.

SUPPLE only supports BLOCK array distributions to exploit locality deriving from stencil computations. Since stencil references are regular and known at compile time, several optimizations such as *message vectorization*, *coalescing* and *aggregation* [5] can be carried out to reduce communication overheads. These optimizations combine element messages which are then sent to the same processor in one single message without redundant data. Another optimization technique implemented in SUPPLE is *iteration reordering*. The local loops executed by each processor according to the BLOCK distribution and the owner computes rule are split into multiple localized loops: those accessing only local data, and those accessing remote data as well. The loops accessing only local data are scheduled between the asynchronous *sends* and the corresponding *receives* which are used to transmit remote data: in this way the execution of the iterations accessing only local data allows communication latencies to be hidden [5]. As far as our flame simulation benchmark (see Fig. 1), we used this scheme to implement the first Convection loop, which is uniform since the cost of the function $F()$ is constant, and exploits a statically predictable and regular *Five-Point stencil* to access the array `B(N,N)`. For the sake of brevity, we will not discuss in more depth the features of SUPPLE for the support of these kind of uniform and regular loops.

The most innovative feature of SUPPLE is, on the other hand, its non–uniform loop support, which adopts a hybrid (static + dynamic) strategy to schedule loop iterations. In the following of this section we describe this part of the SUPPLE support, and its use for the implementation of the Reaction phase of our benchmark.

### 2.1 SUPPLE implementation of the Reaction loop

The innovative feature of SUPPLE is its support for non–uniform loops, such as the one that implement the Reaction phase in our flame simulation code. In this case a hybrid (static + dynamic) scheduling strategy is adopted. At the beginning, to reduce overheads, iterations are scheduled statically, by sequentially executing those stored in a queue $Q$, hereinafter called *local* queue. Once a processor understands that its local queue is *becoming empty*, the dynamic part of the scheduling policy is initiated. Computations are thus migrated at run-time to balance the workload.

Since SUPPLE groups the iterations into *chunks*, the iteration space is statically *tiled*. Chunking is introduced to reduce overhead because SUPPLE migrates chunks instead of single iterations on the basis of the workload distribution, and each processor makes decisions about work migration between the execution of subsequent chunks. To improve locality exploitation [8], SUPPLE adopts tiling also to implement uniform parallel loops.
A further motivation for the adoption of chunking is to avoid synchronizations being introduced between different loop executions. One synchronization might be needed, in fact, if the same array is modified during a previous non–uniform loop, and, due to dynamic work migration, the update of the local array partitions is partially performed by processors that are not the owners of the partitions. SUPPLE avoids waiting for the complete coherence of each data partitions owned by each processor before starting a new loop execution. In fact, it associates 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. Flags are checked only when the corresponding tiles need to be read, thus allowing useful computations to be overlapped with communications. As far our flame simulation kernel code, a flag is thus associated with each tile of the array `C`. In fact, since `C` is updated during the non–uniform Reaction loop (whose implementation exploits a hybrid scheduling of chunks), and is read during the next execution of the uniform Convection loop, the latter loop must check flags for data coherence.

Work migration requires both chunk identifiers and corresponding data tiles to be transmitted over the interconnection network. Migrated chunks, previously stored in the local queue $Q$ of the owner, are stored by each receiving processor in a queue $RQ$, called *remote*. Data tiles, once updated remotely, must be returned to owners. In order to overlap computation with communication latency, and to avoid processors becoming idle while waiting for further work, an aggressive *prefetching policy* is adopted.

The size $g$ of chunks is fixed and is decided statically. This size gives only a lower bound on the amount of work migrated at run-time, because the support adopts a heuristics to decide at run-time how many chunks must be moved from an overloaded processor to a underloaded one. On the other hand, since SUPPLE uses a *polling* technique to probe message arrivals, $g$ directly influences the time elapsed between two consecutive *polls*.

Differently from other proposals [10], SUPPLE dynamic scheduling algorithm is fully distributed and asynchronous, and does not introduce bottlenecks which may jeopardize the efficiency when many processors are used. Chunk migration decisions are taken on the basis of local information only. It 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 [7, 15]. According to the framework proposed by Willebeek-LeMair and Reeves [15], our load balancing technique can be characterized by considering the strategies exploited for *processor load evaluation*, *load balancing profitability determination*, *task migration*, and *task selection*.

*Processor Load Evaluation.* Each load balancer policy requires a reliable workload estimation to detect load imbalances and take, hopefully, correct decisions about workload migration. Our technique is based on the assumption that each processor can derive the expected, average execution cost of its own chunks stored in the local queue $Q$. Since the first part of our technique is static, at the beginning each processor only executes chunks belonging to $Q$. During this phase, the average chunk cost is derived through a non–intrusive code instrumentation. Moreover, when chunks are migrated toward underloaded processors, an estimate of their cost is communicated as well. Each processor is thus aware of the average costs of remote chunks stored in $RQ$. On the basis of the cost estimates of chunks still stored in $Q$ and $RQ$, each processor can therefore measure its own *current load*.

*Load Balancing Profitability Determination.* This strategy is used to evaluate the potential speedup obtainable through load migration weighted against the consequent overheads. In SUPPLE, each processor detects a possible load imbalance, and starts the dynamic part of the scheduling technique on the basis of local information only. It compares the value of its *current load* with a machine-dependent *Threshold*. When the estimated local load becomes lower than *Threshold*, the processor begins to ask other processors for remote chunks. The same comparison of the *current load* with *Threshold* is used to decide whether a chunk migration request should be granted or not. A processor $p_j$, which receives the migration request from a processor $p_i$, will grant the request only if its *current load* is higher than *Threshold*. Note that the *Threshold* parameter is chosen high enough to *prefetch* remote chunks, thus avoiding underloaded processors becoming idle while waiting for chunk migrations.

*Task Migration Strategy.* Sources and destinations for task migration are determined. In our case, since the load balancing technique is receiver initiated, the source (*sender*) of a load migration is determined by the destination (*receiver*) of the chunks. The *receiver* selects the processor to be asked for further chunks by using a *round-robin* policy. This criterion was chosen for its simplicity and also because tests showed that it was the best policy to adopt [7].
SUPPLE uses the same *Threshold* parameter mentioned above to reduce overheads of our task migration strategy. Overheads should derive from requests for remote chunks which cannot be served because the *current load* of the partner that has been asked for is too low (i.e., it is lower than *Threshold*). Thus each processor, when its *current load* (which, in this case, only derives from the chunks in $Q$) becomes lower than *Threshold*, broadcasts a so-called *termination message*. Our round-robin strategy can thus select a processor from those that have not yet communicated their termination.
Termination messages are also needed to end the execution of a parallel loop, and this is also the reason for the name we have given them. When a processor has already received a termination message from all the other processors, and both its queues $Q$ and $RQ$ are empty, then it can locally terminate the parallel loop.

*Task Selection Strategy.* Source processors select the most suitable tasks for effectively balancing the load, and send them to the destinations that asked for load migration. In terms of our support, this means choosing the most appropriate number of chunks that must be moved to grant a given migration request. We use a modified *Factoring* scheme to determine this number [6]. *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 from a central queue, and improves other techniques previously proposed [11, 14]. 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.

## 3 Experimental results

We tested SUPPLE on a 64 node Cray-T3D by using our flame simulation benchmark. The message passing layer used by SUPPLE is MPI, and for the experiments we exploited the MPI library, release CRI/EPCC 1.3, developed by the Edinburgh EPCC in collaboration with Cray Research Inc. We also compared the SUPPLE implementation of the benchmark with an HPF-style one.

### 3.1 The synthetic input data sets

We adopted a set of synthetic data sets because of the need to validate SUPPLE as a general support. Conversely, if we tested SUPPLE on a single problem instance, we might have come to the wrong conclusions due to the specific features of the data set chosen for the test. According to the code skeleton shown in Fig. 1, to build the different data sets we made the following assumptions:

- the simulation grid is $1024 \times 1024$ points (represented by doubles). Due to the data distribution adopted, a block of $128 \times 128$ contiguous points is thus assigned to each of the 64 processing nodes of the target machine;
- one quarter of the total execution time is spent in the Convection phase, while the Reaction phase takes the remaining three quarters;
- experiments were conducted for several values of $\mu$, where $\mu$ is the average execution time to process a grid point during the Reaction phase. Note that, because of the previous assumption about the total execution time, the average execution time per point during the Convection phase was $\frac{\mu}{3}$;
- each input data set was built according to a simple model of load imbalance [9] discussed in the following. Note that the imbalance features of the data set are important for the behavior of the Reaction phase, where the per-point execution time depends on the value of the matrix element processed.

*Model of load imbalance.* Given $\mu$, the average execution time, our model of load imbalance assumes that the workload is not distributed uniformly, and that there exists a region of the simulation grid, hereafter called *loaded region*, in which the average per-point execution time is greater than $\mu$.
The load imbalance model is thus based on two parameters, $d$ and $t$. The parameter $d$, $0 < d \leq 1$, is a fraction of the whole simulation grid, and determines the dimension of the loaded region. The parameter $t$, $0 < t \leq 1$, is a fraction of the total workload $T$, and determines the whole workload concentrated on the loaded region. Therefore, it follows that $t \geq d$.

Note that the imbalance is directly proportional to $t$ for a given value of $d$, since larger values of $t$ correspond to larger fractions of $T$ concentrated on the loaded region. Similarly, the imbalance is inversely proportional to $d$ for a given value of $t$. From these remarks, we can derive $F$, called *factor of imbalance*, defined as $F = \frac{t}{d}$

*Sample data sets.* The data sets used for the experiments were characterized by values of $F$ ranging from 1 to 9, where the size of the loaded region was kept fixed ($d = 0.1$), and $F = 1$ corresponds to the balanced case. All the results reported in this paper were obtained by also keeping fixed the position of the loaded region during all the simulation time steps. We also carried out some tests in which the position of the loaded region is moved at each step. The difference between the results of these experiments and those reported in the paper are not appreciable, because SUPPLE does not exploit any previous knowledge about which the overloaded or the underloaded processors are.

### 3.2 The experimental parameters

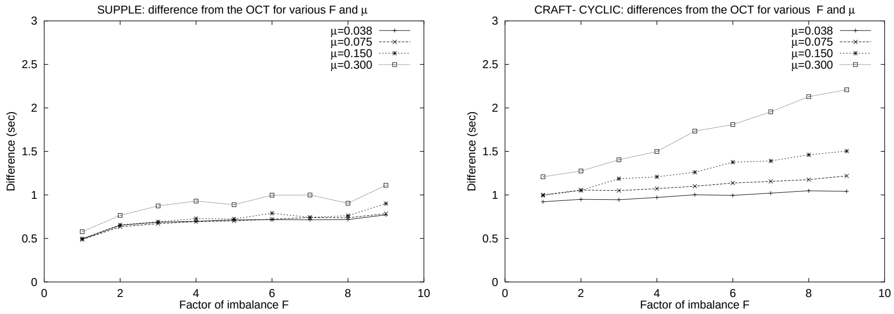
The experiments were conducted on 64 processors for different values of $F$ and $\mu$, and specifically for $F = 1, \cdots, 9$, and $\mu = 0.038,\ 0.075,\ 0.15,\ 0.3\ msec$. As an example of the imbalance introduced by our synthetic data sets during the Reaction phase, consider that, for $F = 9$ and $\mu = 0.3\ msec$, the computation of a single point takes $0.033\ msec$ if the point itself belongs to the unloaded region, while it takes $2.697\ msec$ if it belongs to the loaded region. The external sequential loop of the benchmark was iterated for 10 times in all the experiments.

Given the assumptions above, it is easy to determine the theoretical *Optimal Completion Time* (OCT) when the simulation code is run on 64 processors. The OCT values for $\mu = 0.038, 0.075, 0.15$, or $0.3\ msec$ are 8.2, 16.4, 32.8, or 65.5 $sec$, respectively. This time has been determined without considering any overheads, by symply 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 ($128 \times 128$), times the number of simulation time steps (10).

The only parameters that can be modified and tuned in our support are the size $g$ of a chunk of iteration, i.e. our scheduling unit, and the prefetching $Threshold$. The parameter $g$ affects the scheduling overhead, and the polling mechanism used to check for message arrivals. More specifically, a larger $g$ reduces the scheduling overhead, but a larger $g$ may also reduce the ability of our support to effectively balance the workload. A tradeoff must be found, and this depends on both the architecture and the features of the problem.

The tuning of the other parameter, $Threshold$, mainly depends on the specific architecture. It should be larger on architectures where communications are not very efficient. We found, however, that it is also related to the granularity of each chunk, which in turn depends on $g$ but also on the features of the specific problem, i.e. on the parameters $\mu$ and $F$.

Since $g$ and $Threshold$ are highly interdependent, one strategy is to fix one and tune the other parameter on the basis of the first setting. Hence, all the experiments were carried out by fixing $g = 21$. We changed the granularity of each chunk by using data sets characterized by different $\mu$. We observed that for larger




(a) (b)

**Fig. 2.** Differences from the OCT for various $\mu$ as a function of the factor of imbalance $F$: (a) SUPPLE results, (b) CRAFT Fortran results.

$\mu$, it is better to increase the value of $Threshold$. This is not surprising, since for large chunk granularities, overloaded processors become lazier in answering incoming requests due to our polling implementation of input message checking. Thus, for larger $\mu$, work prefetching should be begun earlier to avoid emptying both $Q$ and $RQ$. To this end, we have to increase the value of $Threshold$. In the exeperiment we used values of $Threshold$ ranging from 2 to 40 $msec$, where larger $Threshold$ values were used for larger $\mu$.

### 3.3 Result assessment

Fig. 2.(a) shows several curves. Each curve is related to a distinct $\mu$, and plots the difference between the completion time of the slowest processor and the theoretical OCT for various factors of load imbalance $F$. The difference from OCT 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 the messages that implement the stencil communications of the first Convection loop. Note that there is an overhead of almost 0.5 $sec$ even in the balanced case, and the overhead becomes larger when the input data set introduces a more extensive load imbalance. For $F = 9$ and $\mu = 0.15$ $msec$, SUPPLE balances the load with a completion time which is very close to the OCT: the difference from OCT is less than 1 $sec$. Note that, in the absence of load balancing actions, the completion time should have been, at least 229 $sec$, with a difference from the OCT of more than 196 $sec$. If we look at the same results by considering, instead of the absolute differences from OCT, their percentage with respect to OCT, then this percentage ranges from 6% to 9.4% for $\mu = 0.038$, while it ranges between 0.88% and 1.69% for $\mu = 0.3$.

### 3.4 Comparison with a HPF-like implementation

We implemented the same benchmark with CRAFT Fortran (an HPF-like language), by feeding it with the same synthetic data sets. Since, in this case, the scheduling is static and depends on the data layout, the simulation grid may be distributed CYCLIC, thus losing data locality in favor of a better balance of the processor workloads. Alternatively, we may adopt a BLOCK distribution for the first Convection loop, and a CYCLIC one for the second Reaction one. This solution does not loose data locality exploited by the Convection loop, but would entail using an executable redistribution directive. Although CRAFT Fortran does not support this directive, we simulated it by assigning one array distributed CYCLIC to another distributed BLOCK, and vice versa. Finally, to find a tradeoff, some combinations of BLOCK and CYCLIC may be adopted.

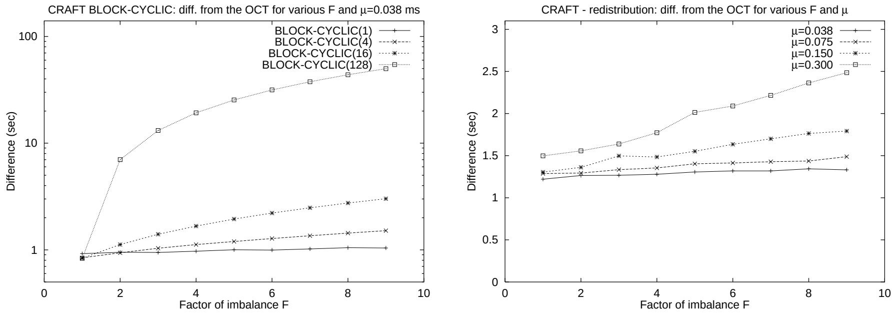
The best results for unbalanced problem instances were obtained by using a pure CYCLIC distribution (see Fig. 2.(b)). One of the reasons of these performance results is the computational weight of the Reaction loop, which is three times that of the first Convection loop. Thus it is more important to try to balance the workload through a pure CYCLIC distribution than to increase locality exploitation with a BLOCK distribution. The differences from the OCT are larger than those obtained by SUPPLE, even for the completely balanced case (nearly a half second more than SUPPLE). These bad results for $F = 1$ are also due to the CYCLIC distribution, which should be turned to BLOCK to take advantage of data locality in the first Convection loop. The results become worse for larger $\mu$ and $F$. For example, for $F = 9$, the CRAFT differences from the optimal times are 1.04, 1.22, 1.50, 2.21 *sec* for $\mu = 0.038$, 0.075, 0.15, 0.3 *msec* respectively, against the 0.77, 0.78, 0.90, 1.11 *sec* obtained on the same tests by SUPPLE. The CYCLIC distribution, besides losing data locality, is not able to perfectly balance the workload, so that, for larger $\mu$, the workload differences between overloaded and underloaded processors increase.

We also conducted other tests with CRAFT Fortran to evaluate combinations of BLOCK and CYCLIC distributions. These results were worse than the pure CYCLIC case. Fig. 3.(a) refers to the case $\mu = 0.038$ *msec*, and shows some curves that plots the differences from the OCT for different values of $F$. Note that the ordinate scale is logarithmic, and that the pure BLOCK distribution is identified by the label BLOCK_CYCLIC(128).

Finally, we tested the redistribution of the simulation grid between consecutive parallel loop executions. The results, which are shown in Fig. 3.(b) are worse than those obtained by using a pure CYCLIC distribution for both loops. The reason may be the huge volume of data moved simultaneously between all processors, which may cause network congestion problems.

## 4 Conclusions

We have described SUPPLE, an innovative run–time support, which can be used to compile either uniform or non–uniform parallel loops with regular stencil data references. SUPPLE exploits regular BLOCK distributions of the arrays involved to favor data locality exploitation, and uses the static knowledge of the data layout and of the regular stencil references to perform well-known optimizations such as *message vectorization*, *coalescing* and *aggregation*. *Iteration reordering*




(a) (b)

**Fig. 3.** CRAFT Fortran differences from the OCT: (a) for different data layouts with $\mu = 0.038$ *msec*, (b) for different $\mu$ with array redistribution.

is exploited to hide the latencies of the communications needed to fetch remote data. SUPPLE supports two different loop scheduling strategies. Parallel loops characterized by iterations whose costs are uniform, can be scheduled according to a pure static strategy based on the *owner computes rule*. A hybrid scheduling strategy that tries to balance processor workloads can be used instead if the load assigned to the processors according to the owner computes rule is presumed to be non–uniform. In this case loop iterations are initially scheduled statically, and, if a load imbalance is actually detected, an efficient dynamic scheduling, which requires data tiles and iteration indexes to be migrated, is started.

We have shown that SUPPLE hides most dynamic scheduling overheads produced by the need for monitoring the load and moving data, by overlapping them with useful computations. To this end it exploits an aggressive prefetching technique that tries to avoid underloaded processors from becoming idle while waiting for workload migration requests to be satisfied. Nevertheless, since data may be migrated and updated remotely due to dynamic scheduling decisions, and the same data must be kept coherent on the original owner, SUPPLE adopts a fully asynchronous coherence protocol that allows useful computations to be executed while waiting for data to become coherent.

The results of our experiments conducted on a Cray T3D machine by running a kernel code of a flame simulation application, have shown that the performances obtained are very close to the optimal ones. To validate SUPPLE as a general support for non–uniform loops 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 workload imbalances. For data sets characterized by an average iteration execution times of 0.3 *msec*, the overheads due to communications and the residual load imbalance range between 0.88% and 1.69% of the optimal time. In addition, we have compared the SUPPLE results with those obtained running a CRAFT Fortran implementation of the same benchmark. In all the cases tested, the SUPPLE implementation obtained better performances.

## 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. High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993. Version 1.0.
3. High Performance Fortran Forum. *HPF-2 Scope of Activities and Motivating Applications*, Nov. 1994. Version 0.8.
4. 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.
5. 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.
6. S.F. Hummel, E. Schonberg, and L.E. Flynn. Factoring: A Method for Scheduling Parallel Loops. *Comm. of the ACM*, 35(8):90–101, Aug. 1992.
7. V. 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.
8. 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.
9. 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.
10. 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.
11. C. Polychronopoulos and D.J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Trans. on Computers*, 36(12), Dec. 1987.
12. 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.
13. 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.
14. T.H. Tzen and L.M. Ni. Dynamic Loop Scheduling on Shared-Memory Multiprocessors. In *Proc. of Int. Conf. on Parallel Processing - Vol II*, pages 247–250, 1991.
15. 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.
16. H.S. Zima and B.M. Chapman. Compiling for Distributed-Memory Systems. *Proc. of the IEEE*, pages 264–287, Feb. 1993.