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

Technical Report CS-96-17

December 23, 1996

Salvatore Orlando and Raffaele Perego

SUPPLE: an Efficient Run–Time Support for Non–Uniform Parallel Loops

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

SUPPLE: an Efficient Run–Time Support for Non–Uniform Parallel Loops*

Salvatore Orlando†, Raffaele Perego‡

December 23, 1996

Abstract

The efficient implementation of parallel loops on distributed–memory multicomputers is a hot topic of research. To this end, data parallel languages generally exploit static data layout and static scheduling of iterations. Unfortunately, when iteration execution costs vary considerably and are unpredictable, some processors may be assigned more work than others. Workload imbalance can be mitigated by cyclically distributing data and associated computations. Though this strategy often solves load balance issues, it may worsen data locality exploitation.

This paper presents SUPPLE (SUPort 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. We have compared our results with those obtained with a CRAFT Fortran implementation of the benchmark.

Keywords: Data parallelism, data parallel languages, loop scheduling, load balancing, run–time supports, compiler optimizations.

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) [1], Vienna Fortran [2] and Fortran D [3] 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 [4], 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. In this paper we are interested in run-time supports for parallel loops with regular stencil data

*This paper has been submitted for publication to IEEE Transactions on Parallel and Distributed Systems.

†S. Orlando is with the 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

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

references. In particular, we consider *non–uniform parallel loops*, i.e. parallel loops in which the execution time of each iteration varies considerable and cannot be predicted statically.

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. Arrays are distributed according to the directives supplied by programmers, and computations, i.e. the various loop iterations, are assigned to processors following a given rule which depends on the data layout. The most commonly used rule is the *owner computes rule*, according to which the various assignment statements appearing in each iteration of the loop are mapped to the processor that owns the array data item which is updated.
HPF allows only regular data layouts to be specified. This depends on the limited possibilities a programmer has to express data mapping. Arrays can be distributed by means of the BLOCK directive which assign contiguous array elements to the local memory of the same processor, or by the CYCLIC directive to scatter data to the local memories of the various processors. Since the assignment of computations derives from the chosen data layout, a BLOCK distribution is usually adopted to exploit data locality: computations mapped on a given processor by the compiler mainly use the data block allocated to the corresponding local memory. Conversely, CYCLIC distribution is usually adopted when load balancing issues are more important than locality exploitation. A combination of both the distributions, where smaller array blocks are scattered on processing nodes, can be adopted to find a tradeoff between locality exploitation and load balancing. It is worth noting, however, that the choice of the best distribution is still up to programmers, and that the right choice depends on the features of the particular application. The general problem of finding an optimal data layout is in fact in NP [5].

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 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 [6]. The techniques proposed are mainly based on run-time codes which collect information during the first phases of computation, and then use this information to optimize the execution of the following phases of the same computation. An example of these techniques is the one adopted by the CHAOS support [7] to implement non–uniform parallel loops. The idea behind this feature of the CHAOS library is the run–time *redistribution* of arrays, and the consequent re–mapping of iterations. Redistribution is carried out synchronously between the execution of subsequent executions of parallel loops, and is decided on the basis of information (mainly, timing information) collected at run–time during a previous execution of the loop. If the original load distribution is not uniform, the data layout is thus modified in order to balance the processor loads. Note that, according to the HPF standard, programmers should be allowed to redistribute arrays at run-time, but, due to the limited distribution capabilities of HPF, also the new data layout must be regular. In [8] a dialect of HPF has been extended with new language constructs which are interfaced with the CHAOS library to support irregular computations.

In this paper we address non–uniform parallel loop implementations, introducing a truly innovative support called SUPPLE (SUPort for Parallel Loop Execution). 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 [9]. 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 support that does not introduce barriers between consecutive iterations of loops. Below we describe our support in detail, and we show some results obtained by running the kernel of a flame simulation code on a Cray T3D. In this application, the

```
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.

characteristics of the input data set strongly influence the time needed to compute loop iterations and thus may cause workload imbalances. For this reason, we ran our tests on synthetic data sets, built on the basis of a simple load imbalance model. To compare SUPPLE with an HPF-style static approach, we used the Cray CRAFT-Fortran, whose language directives and compilers were partially adapted from Fortran-D [3] and Vienna Fortran [2].

The paper is organized as follows. Section 2 describes the flame simulation code we used as a benchmark to validate the SUPPLE approach. Section 3 presents the main features of SUPPLE and gives some details on its implementation. Assessments of the experiment results are reported and discussed in depth in Section 4, which also describes the load imbalance model used to build the synthetic data sets. Finally, Section 5 provides a summary of related work, and Section 6 draws the conclusions.

2 The benchmark

The benchmark we used to validate the SUPPLE approach is directly derived from a class of actual scientific codes used to carry out detailed flame simulations [10]. 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. 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 detail in [8]. It is also included in the HPF–2 draft [6] as one of its motivating application codes. It 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 workload distribution evolves as the simulation progresses [8].

The HPF-like code of a simplified two-dimensional flame modeling code is reported in Fig. 1. The 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 local memories. Unfortunately, since the workload distribution evolves and depends on the specific features of the input data set, we cannot devise at compile time a distribution that is suitable for both phases of the simulation. The following are therefore possible solutions.

1. CYCLIC distribution of the simulation grid thus losing data locality in favor of a better balance of the processor workloads. This solution must be followed to obtain acceptable performances with HPF-like data-parallel languages, which currently do not support irregular applications [6]. A combination of BLOCK and CYCLIC distribution may also be used to find a tradeoff between locality exploitation and load balancing. However, static optimizations like these may be unsuccessful when the workload distribution varies unpredictably at run time.

2. Using BLOCK distribution for the first loop, and CYCLIC for the second. This solution would entail using an executable redistribution directive. Although several HPF versions do not support this directive, we can simulate it by assigning one array distributed CYCLIC to another distributed BLOCK, and vice versa.

3. BLOCK partitioning the grid in the first convection phase, and then suitably redistributing data and iterations to balance the workloads before starting the reaction computation. This approach is different from the previous one, because data and computation reallocation is minimized, and is carried out at run time on the basis of information collected during previous phases of the computation. This implementation scheme is followed by Ponnusamy *et al.* in [10, 8]. They exploit the CHAOS run–time library [7], whose functionalities are used to build at run-time an irregular data redistribution plan resulting in a nearly balanced reaction computation. An array `wt[]`, aligned with the simulation grid, stores at time step t the costs of reaction calculation at each grid point. A bin-packing heuristic on `wt[]` is then used at iteration $t + 1$ to obtain a good irregular distribution for the data structures involved. Data are recombined in regular blocks before beginning the convection phase of time step $t + 2$. Redistribution for achieving load balance takes about 15-20% of the total execution time, but results in execution times 4–10 times better than without load balance actions [8]. The authors do not compare the approach with the first, much simpler solutions.

4. BLOCK partitioning the grid to exploit data locality, and dynamically redistributing the expensive reaction calculations to balance the load. This is the approach discussed in this paper. SUPPLE uses the *owner computes rule* to statically schedule iterations belonging to the first convection parallel loop, and exploits a hybrid strategy for the reaction calculations. Non–uniform loops are executed by adopting, as far as possible, a static scheduling policy, and dynamically moving data and iterations among processors only if a load imbalance is detected. SUPPLE builds the balanced schedule from scratch at every loop execution: this is useful to efficiently implement highly adaptive application codes in which the workload varies at each iteration. Note that the technique adopted in CHAOS may not work for this type of code, because the workload distribution is carried out before starting a new loop execution on the basis of information collected during previous executions.

3 Overview of SUPPLE

SUPPLE is a run-time support for the efficient implementation of data parallel loops with regular, *stencil* references [11], 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* [9] 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 assigned to every 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 [9].

```
FORALL (i= 2:99,  j= 2:99)
  A(i,j) = F(B(i,j), B(i-1,j), B(i,j-1), B(i+1,j), B(i,j+1))
END FORALL
```

Figure 2: A simple loop characterized by a *Five-Point Stencil*.

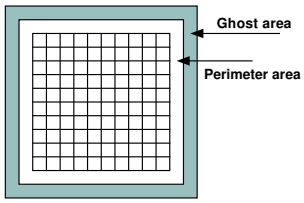


Figure 3: Ghost, perimeter and inner areas of a block partition.

Besides uniform loops, SUPPLE also supports non–uniform loops through the adoption of an innovative dynamic load balancing technique. Dynamic scheduling is combined with the optimized static scheduling illustrated above. This hybrid (static + dynamic) strategy only applies to loop iterations that access local data. Loop iterations that need to fetch remote data are simply scheduled according to the owner computes rule. This limitation in the dynamic scheduling is not very strong, and does not affect the ability of our technique to balance the overall workload. Consider, in fact, that the number of iterations that access remote data is usually a small fraction of the total when data sets of interesting sizes are BLOCK distributed and stencil computations apply. One of the reasons for this limitation is to avoid the introduction of irregularities in the stencil data references, thus preventing the possibility of optimizing communications.
The hybrid scheduling strategy works as follows: at the beginning iterations are scheduled statically according to the optimized ordering mentioned above, and start being dynamically migrated only if a load imbalance is actually detected at run-time. The load balancing heuristic adopts local policies of imbalance detection to avoid global synchronizations, and prefetching of remote iterations to hide communication latencies.

This section presents SUPPLE in detail by considering a simple example: the implementation on a processor grid of a parallel loop accessing bidimensional arrays. In particular, we describe the optimized implementation of stencil communications, and both the static and hybrid (static + dynamic) scheduling strategies that can be adopted. SUPPLE supports, however, different processor layouts, multi-dimensional arrays distributed in any dimensions, and parallel loops with different stencil references. To this end, SUPPLE uses several descriptors to store information about the arrays and loops involved, and provides routines to fill the descriptor fields used at run-time by the implementation code.

3.1 Management of stencil references

SUPPLE provides the support to implement uniform loops by adopting the scheduling and the communication optimizations that can be employed when data are accessed with regular stencils [9, 11]. To simplify the presentation, we consider the simple parallel loop shown in Fig. 2, and characterized by a regular *Five-Point stencil* used to reference the input array `B(100,100)`. We discuss the case in which the arrays `A(100,100)` and `B(100,100)` are aligned and distributed in square blocks on a 4×4 processor grid. We distinguish the case of uniform iteration costs from the case in which non–uniform costs on function `F` may introduce a workload imbalance.

Before discussing how stencil references are managed by SUPPLE, we need to introduce some terminology. Given a stencil, for example the *Five-Point* stencil $\{(i,j),\ (i+1,j),\ (i-1,j),\ (i,j+1),\ (i,j-1)\}$ of the loop in Fig. 2, we can define (i,j), i.e. the loop indexes of the generic iteration, as the *seed* of the stencil. For each dimension and for each stencil point, we can compute the *distance* from the seed. For example, if we consider the first dimension, the distance of the stencil point (i',j') from the seed (i,j) is given by the

```
CALL PACK_PERIMETER(Local_B)
CALL SEND_TO_NEIGHBORS(Local_B)
CALL RECEIVE_FROM_NEIGHBORS(Local_B)
CALL UNPACK_GHOST(Local_B)
DO (i= FROM1:TO1,  j= FROM2:TO2)
  Local_A(i,j) = F(Local_B(i-1,j-1), Local_B(i-2,j-1), Local_B(i-1,j-2),
                   Local_B(i,j-1), Local_B(i-1,j))
END DO
```

Figure 4: Localized loop executed by a generic processor before iteration reordering.

expression $|i' - i|$. Finally, we can define the *width* of the stencil as the maximum of all the distances for all points and dimensions. This width is equal to one for the Five-Point stencil used in the code of Fig. 2.

Thus, given a stencil and its width, we allocate on each processing node enough memory to host the array partition assigned according to the BLOCK distribution, and a surrounding *ghost* (or *overlap*) area (see Fig. 3). The *ghost* area has the same width as the stencil and is used to buffer the *perimeter* areas of the adjacent partitions owned by neighboring processors, and accessed through non local references. With respect to the example reported in Fig. 2, a 26×26 local array is thus assigned to each processor, since we have to consider the 1-point-wide ghost area which surrounds the original 25×25 block. At this point a trivial implementation of the parallel loop can be derived. An equal fraction of loop iterations is assigned to each processor according to the owner computes rule, and the loops executed by each processor have to be *localized*, i.e. their boundaries and array references must reference to the local array block. Before executing the loop, however, non local data must be fetched from the nearest neighboring nodes. This means that explicit communications have to be inserted before the localized loop. In particular, first each processor has to pack data belonging to its *perimeter* area (see Fig. 3) into suitable messages, and send them to its neighbors. Then each processor waits for the corresponding messages sent by its neighbors and, once received, unpacks them and stores the received data in the ghost area. SUPPLE optimizes the communications required to implement stencil computations by avoiding sending several messages or replicated data to the same processor, i.e. by exploiting message vectorization, coalescing and aggregation optimizations [9]. Fig. 4 shows the simple localized loop executed by a generic processor. Note the references to the local partitions of the whole arrays and the variables that determine the loop boundaries, and that are assigned by specific SUPPLE initialization routines.

The simple implementation outlined above does not hide communication latency. We have to wait for communication completion before starting the execution of the parallel loop. To overlap communications with useful computations, SUPPLE exploits iteration reordering [9]. If we look at Fig. 3 and at the iterations executed by the localized loop of Fig. 4, we can distinguish those assigning the perimeter area and those assigning the remaining *inner* area of the array blocks. Loop iterations assigning the inner area only reference local data, while the others, which also access the ghost area, need to wait for non local data. Hence, we can reorder the loop iterations by executing the iterations that update the inner area between the asynchronous sending of messages and the corresponding receiving. The localized loop in Fig. 4 is thus split into two parts, thus updating different areas. The code which implements the parallel loop thus has four sequential steps:

```
1.  < pack and send perimeter areas to neighboring processors >
2.  < execute localized loops that update inner areas >
3.  < receive from neighboring processors and unpack ghost areas >
4.  < execute localized loops that update border areas >
```

Steps 2 and 4 include the execution of the new parallel loops obtained by iteration reordering, while steps 1 and 3 correspond to communications that implement stencil data references. Steps 1 and 3 must not be executed if the original loop only references local data. Nevertheless, if a parallel loop accesses several arrays, steps 1 and 3 only exchange data for those arrays for which stencil references apply.

Note that in step 4 we used the term *border* rather than perimeter. If we consider the implementation of a single loop like that in Fig. 2 and the array `B`, i.e. the array that is accessed through stencil data references, the border area and perimeter area of the array are identical. The term perimeter, however, refers to a

```
Static_Scheduler (queue Q)
T: chunk;
begin
  foreach T ∈ Q
    while (* the full/empty flag associated with T is set *) do
      Wait_for_coherency(T);
    end while
    Execute (T);
  end foreach
end
```

Figure 5: Pseudo-code of step 2 when static scheduling is adopted.

memory area that stores a part of a specific array, array `B` in this case, while border refers to the space indexes of the localized loop. Note that if we consider array `A`, its perimeter and ghost area widths are equal to zero, because the parallel loop does not contain stencil references on `A`. Despite this, array `A` is updated by following the same schedule: step 2 updates the inner area, and then step 4 updates the borders.
A simple per-loop strategy to select the width of border area is thus the following: choose a border width that is equal to (or larger than) the width of the largest stencil among those used to access all the arrays referenced in the loop. To obtain more effective optimizations, however, SUPPLE requires compilers (or programmers) to analyze all the parallel loops that access the same array. This leads to a choice of a border width that may be larger than the one chosen by the previous strategy. The goal of the new strategy is to avoid different parallel loops updating the same array with different iteration ordering schemas, and it is motivated, as we will see below, by the hybrid scheduling technique employed to implement non–uniform loops.

While step 4 is straightforward, because it simply requires the execution of the iterations which modify data elements belonging to the border area, step 2 is more complex, because a hybrid scheduling technique (static + dynamic) may be adopted if the loop is supposed to be non–uniform. In other words, SUPPLE may be used to exploit either a simple static scheduling or a more complex hybrid one. To support dynamic scheduling, the iterations that assign the elements lying on the inner area shown in Fig. 3 are statically grouped into *chunks* by *tiling* the iteration space. Tiling is necessary because SUPPLE migrates chunks of iterations on the basis of the workload distribution. Note that Fig. 3 shows the tiling of the inner area, which appears to be subdivided into further small sub-blocks.
Tiling is not strictly needed if static scheduling is exploited, but we adopted it in this case too because it has been proved that it may improve locality exploitation [12]. On the other hand, when the loop is dynamically scheduled, tiling helps to avoid synchronizations being introduced between different loop executions. One synchronization might be needed 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. A possible synchronous solution would be to wait, at the end of the non–uniform loop, for the arrivals of all the messages that contain the updates generated by remote processors. These messages are needed to restore the coherence of the data partitions owned by the various processors. SUPPLE avoids these synchronizations, because it associates a *full/empty-like* flag [13] 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. The implementation of this asynchronous coherence protocol is the only reason to keep uniform the dimensions of border and inner areas between different related loops. In fact, this allows us to maintain uniform the shape and the number of data tiles accessed within different parallel loops.

Fig. 5 shows the SPMD pseudo code that implements step 2 of the general technique described above. All the code is included in the routine *Static_Scheduler()*, which statically schedules the chunks of iterations by accessing local data only. The queue Q contains this chunk sequence.
Note the subroutine *Wait_for_coherency()*, which guarantees the correct program semantics by waiting until the data tile associated with T becomes coherent, and the subroutine *Execute (T)*, which is a simple local-

ized loop that updates the associated data tile. The results discussed in Section 4 show that the routine *Wait_for_coherency(T)* does not actually block computation because, when it is invoked, the corresponding coherence message has already been received. Note that, however, the **while** loop which waits for coherency can be completely removed if the data tiles of the arrays involved are not scheduled dynamically by other parallel loops. As far the benchmark illustrated in Section 2 is concerned, we have used such a static scheduling scheme to implement the uniform *Convection* loop (see Fig. 1). The width of the border area is in this case equal to one, while a ghost area of the same width surrounds the blocks of array `B`. The resulting inner area is tiled, and the chunks are scheduled statically. Each chunk updates the array `A` and reads both arrays `B` and `C`. Before the execution of each chunk, according to the code in Fig. 5, we wait for the coherence of the corresponding tile of array `C`. Coherence is checked because the array `C` is updated by the non–uniform *Reaction* loop, which is implemented by adopting the hybrid scheduling scheme described in the next section.

3.2 Non–uniform loop implementation

The innovative feature of SUPPLE is its support for non–uniform loops, which is attained by adopting a hybrid (static + dynamic) scheduling strategy to implement step 2 of the general technique described in the previous section. At the beginning, to reduce overheads, iterations are scheduled statically, by sequentially executing the chunks stored in queue Q, hereinafter called *local* queue. Once a processor understands that its local queue is *becoming empty*, the dynamic part of our scheduling policy is initiated. Computations are migrated at run-time to balance the workload. Work migration requires both chunk identifiers and corresponding data tiles to be transmitted over the interconnection network. If loops access an array with a given stencil data references, tiles with a suitable surrounding area must be migrated. This is exactly the case of a parallel loop like the one shown in Fig. 2 where the cost of function `F` is variable. In order to overlap computation with communication, an aggressive prefetching policy is adopted to avoid processors becoming idle while waiting for further work. Migrated chunks are stored by each receiving processor in a queue RQ, called *remote*.

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*. We will discuss these issues in detail in Section 4.

The dynamic scheduling algorithm is fully distributed and asynchronous. 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 [14, 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. The architecture we have used, a Cray T3D, is based on the Digital Alpha processor, which provides an inexpensive machine instruction to read the clock cycle count, thus reducing the overhead of 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. As discussed in the paper by Kumar et al. [14], it is more important not to spend too much time in making a decision than always making the best decision.
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* 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: in fact, no executable chunks can be fetched from any remote processors and from both queues. Note that, when a processor ends its parallel loop execution, it has certainly already broadcast its termination message.

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 [16]. *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 storing 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.

3.2.1 SPMD code of the load balancer.

Fig. 6 shows the SPMD pseudo code to implement step 2 of the general technique above. The code exploits the hybrid (scheduling + dynamic) scheduling used by SUPPLE to implement non–uniform parallel loops.

All the code is included in the routine *Hybrid_Scheduler()*, which schedules the iteration chunks of the local Q either locally, according to a static sequential ordering, or remotely, according to dynamic scheduling decisions made only if load imbalance occurs.

The core part of the code is a **while** construct, which ends looping when the function *Terminated&Empty_queues()* returns a *True* value. This function checks the termination condition: a processor that is executing this SPMD code terminates as soon as its queues Q and QR are empty, and it has received a termination message from all the other processors. The function *My_load()* estimates the current *workload* of a processor. It returns the product of the number of chunks still stored in Q times the average local chunk execution cost, plus the the cost estimated for any chunks stored in RQ. Q chunk average cost is determined by monitoring the execution of the local chunks, while estimated costs of remote chunks are communicated by sender processors along with data. At the beginning, when no chunk executions have been monitored, these costs are initialized with high values to avoid wrong chunk requests and migrations.

The prefetching policy adopted to reduce overheads of chunk migration is driven by a machine-dependent *Threshold* parameter: a processor sends a request for remote chunks by calling subroutine *Prefetch_Chunk()* only if its *current load* is lower than *Threshold*. Likewise, the processor that has been asked for, grants the request only if its own load is greater than *Threshold*. Since latencies of chunk migrations may be very large, we may risk prefetching too much because we might send many requests before the first remote chunks arrive. To avoid this, we fix a limit on the number of requests sent by a processor and not yet granted. *Prefetch_Chunk()* sends a request message only if this limit has still not been reached.
Prefetch_Chunk() employs a simple local *round-robin* policy to select the next processor to be asked for a remote chunk. The next partner is selected from those which have not yet communicated their termination by means of a call to subroutine *Comm_termination()*.
The function *Extract()* returns a chunk from a queue. The subroutine *Execute()* executes a chunk T, and, if T is local, measures the time taken to complete it and consequently updates the estimate average cost of local chunks.

We adopt an asynchronous *coherence protocol* to migrate updated data from the processor that actually executed the chunk to the processor that is the owner of the corresponding data partition. The updated data tiles are transmitted by the subroutine *Send_Results_Coherence()*, which also implements message vectorization by buffering more data tiles to be sent to the same processor. The subroutine inserts data tiles in a buffer, and transmits the buffers only when it becomes full. A *full/empty-like* technique [13] is used by our coherence protocol to avoid processing invalid data. When processor p_i sends a chunk b to p_j, it sets a flag marking the data tile to be modified by p_j as invalid. The next time p_i needs to access the same data, for example during the execution of another loop accessing the same array, p_i checks the flag and, if the flag is still set, waits for the updated data tile from node p_j. We have already seen how this check is performed in the static scheduling code of Fig. 5. The same job is performed in the code of Fig. 6 by the function *Extract()* when applied to the local queue Q.
Such a control of data consistency does not entail stopping waiting for coherence messages at the end of parallel loop execution: a new loop can be started and useful computation can be overlapped with these communications.

Finally, the subroutine *Request_Handler()* performs many of the above tasks. This subroutine whose code, for the sake of simplicity, is not shown in detail, polls and handles all the messages that can arrive at a processor. For example:

- requests for more work from underloaded processors, which entail deciding about migration profitability and possibly extracting a bunch of chunks from Q (according to the *Factoring* heuristics) and sending it to the requesting processor;
- coherence messages transporting updated data tiles, which entail copying the received data in the local array partitions and unsetting the corresponding full/empty flags;
- remote chunk bunches sent by overloaded processors, which have to be inserted into the remote queue QR;
- termination messages which modify the behavior of our task migration strategy and are needed to manage termination detection.

```
Hybrid_Scheduler (queue Q)
RQ: queue; T: chunk;
begin
  Initialize_Data_Structures ();
  while (not Terminated&Empty_queues (Q, QR)) do
    if (My_Load (Q, RQ) < Threshold) then
      Prefetch_Chunk ();
      if (* My_Load (Q) < Threshold for the first time *) then
        Comm_termination ();
      end if
    end if
    T = Extract(Q);
    if (Empty (T)) then
      T = Extract (RQ);
      Execute (T);
      Send_Results_Coherence (T);
    else Execute (T);
    end if
    Request_Handler ();
  end while
end
```

Figure 6: Pseudo-code of step 2 when hybrid scheduling is adopted.

4 Experimental results

We have tested SUPPLE on a 64 node Cray-T3D. The message passing layer used by SUPPLE is MPI, release CRI/EPCC 1.3, developed by the Edinburgh Parallel Computing Centre in collaboration with Cray Research Inc. We have also conducted some tests by exploiting a kernel MPI library built on the top of the Fast Messaging (FM) layer developed at the University of Illinois at Urbana-Champaign [17]. The experiments are concerned with a SUPPLE implementation of the benchmark illustrated in Section 2 on a set of synthetic data sets presented in Section 4.1. We have also compared the SUPPLE implementation with an HPF-style one. For the HPF implementation we have used the Cray CRAFT-Fortran, whose language directives and compiler have been adapted in part from Rice University's Fortran-D project [3] and Vienna Fortran [2].

4.1 The synthetic input data sets

We have 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 make the following assumptions:

- the simulation grid is 1024×1024 points. Since each point is a double precision floating point number, each matrix associated with the simulation grid thus requires 8 MBytes of RAM.

  Due to the data distribution adopted, a block of 128×128 contiguous points is 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;

- μ 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 must be $\frac{\mu}{3}$. We have conducted experiments with input data sets characterized by several values of μ to test SUPPLE for different computational grains;

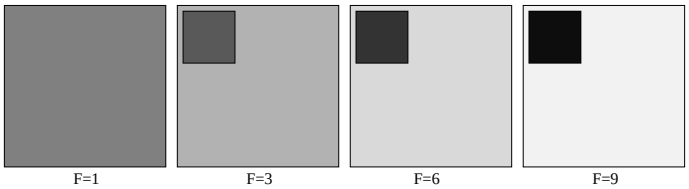


Figure 7: Representation of different simulation grids characterized by different factors of imbalance F.

- each input data set is built according to a simple model of load imbalance [18] 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 μ, 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 μ.

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

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

Sample data sets

Fig. 7 shows a representation of the different data sets we have used to force an unbalanced workload during the reaction phase of our benchmark. We employed data sets characterized by values of F ranging from 1 to 9, where the size of the loaded region is kept fixed ($d = 0.1$). The grey levels used in Fig. 7 represent the workloads associated with grid points, where a darker grey stands for a heavier workload. Note that the case $F = 1$ corresponds to a data set that does not introduce any workload imbalance, since the computational cost of each reaction point simulation is equal to μ.

Fig. 8 shows the blocking distribution of one of our synthetic data sets on a grid of 8×8 processors. Due to the position of the grey loaded region, processors 0, 1, 8, and 9 appear to be overloaded, while processors 2, 10, 16, 17, and 18 are only partially loaded. We will refer to this figure to explain the overheads introduced by our support during the dynamic load balancing phase.

All the results reported in this paper were obtained by keeping fixed the position of the loaded region during all the simulation iterations. We also carried out some tests in which the position of the loaded region is moved at each iteration to simulate the adaptivity of the code. 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.

4.2 The experimental parameters

The experiments were conducted on 64 processors for different values of F and μ, and specifically for $F \in \{1, \cdots, 9\}$, and $\mu \in \{0.038\ msec,\ 0.075\ msec,\ 0.15\ msec,\ 0.3\ msec\}$. The external sequential loop of the flame simulation code was iterated for 10 time steps. Table 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 points belonging to the unloaded and loaded region.

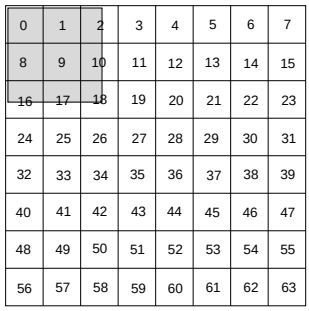


Figure 8: Data layout of a data set characterized by $d = 0.1$ on a 8×8 mesh.

Table 1: Reaction phase: per–point execution time for some values of μ and F

μ	F	*Loaded points*	*Unloaded 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 2 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).

4.3 Tuning the dynamic load balancing parameters

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 since it reduces the number of scheduling units stored in each local queue. On the other hand, a larger g may reduce the ability of our support to effectively balance the workload, mainly because, due to the polling mechanism adopted, it makes the response time of overloaded processors slower when they are asked for by underloaded ones. 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 μ and F.

From the remarks above it follows that the two parameters g and *Threshold* are highly interdependent.

Table 2: Optimal Completion Time (OCT) on 64 processors for different μ

μ	0.038 $msec$	0.075 $msec$	0.15 $msec$	0.3 $msec$
OCT	8.2 sec	16.4 sec	32.8 sec	65.5 sec

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 $g = 21$, so that each chunk assigns the iterations which update a tile of 7×3 points of the simulation grid. The number of chunks scheduled by each parallel loop is thus 756. We changed the granularity of each chunk by using data sets characterized by different μ. We observed that for larger μ, i.e. for chunks with a larger granularity, 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 μ, prefetching further work should be begun earlier to avoid emptying both the local and the remote queues. To do this we have to increase the value of $Threshold$. Hence, we used values of $Threshold$ ranging from 2 to 40 $msec$, where larger $Threshold$ values are used for larger μ. We found, however, that a good heuristic to solve this issue is:

- fix a chunk dimension g, which depends on the size of the problem. The number of chunks must be large enough to allow the load balancing algorithm to work properly: a small number should only allow a few very large bunch of iterations to be moved, thus introducing probable wrong workload balances. Conversely, a very large number of chunks can increase the overhead introduced by our algorithm, though for some applications, we may have cache performance benefits from scheduling loop iterations using small tiles [12];
- start the load balancing algorithm with a small value of $Threshold$, and increase its value at run-time if necessary. When an underloaded processor begins to receive remote chunks along with their expected execution times, on the basis of this value it can increase, if necessary, the current $Threshold$ to take into account the possible lazy behavior of remote overloaded processors. Both the initial value of $Threshold$ and the amount of increase depend on the specific architecture.

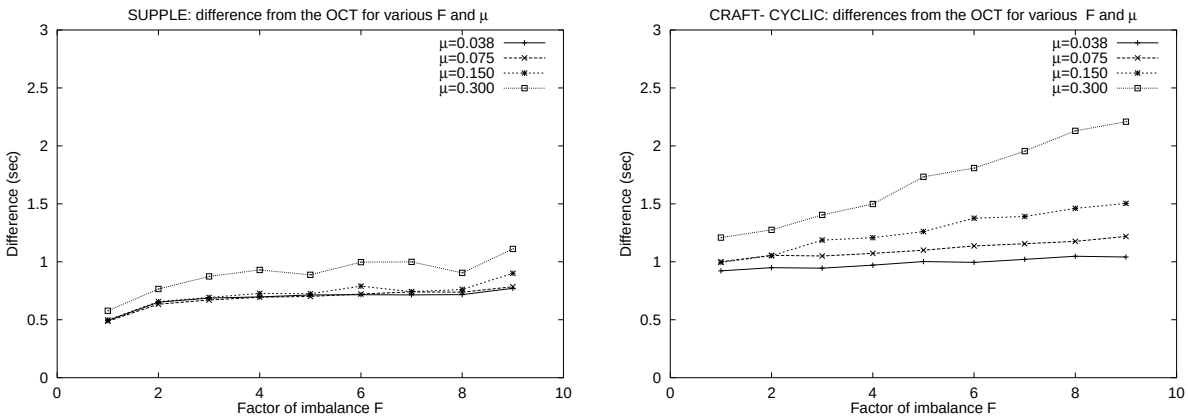
The tests showed that the heuristic above works, with a small decrease in performance with respect to the case in which the correct value of $Threshold$ is established before running the program. However, to understand whether it is possible to avoid this static/dynamic tuning, an implementation of SUPPLE should be tested which uses a different technique to check for message arrivals. For example, a support which exploits interrupts when the polling mechanism starts not working properly [19] and/or active-messages [20] for more efficient message handling.

4.4 Result assessment

Fig. 9.(a) shows several curves. Each curve is related to a distinct μ, and plots the difference between the completion time of the slowest processor and the theoretical OCT (see Table 2) for various factors of load imbalance F. The difference shown in this and in the following plots 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 by implementing the stencil communications of the first convection loop. 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 is nearly 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 their optimality in terms of percentage with respect to the optimum time, then for $\mu = 0.038$ this percentage ranges from 6% to 9.4%, while for $\mu = 0.3$ it ranges between 0.88% and 1.69%.

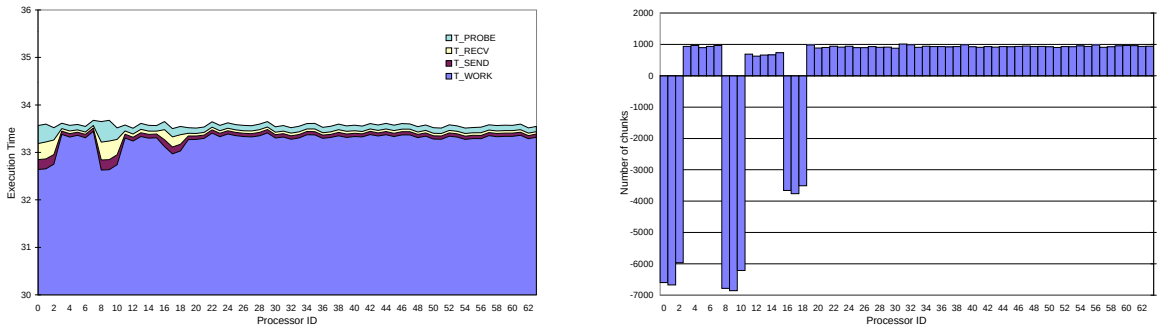
We have carried out some other tests by adopting an MPI kernel build on the top of the FM messaging layer [17]. More specifically, we used the **Pull FM** layer, which improves performance in the presence of a network traffic which may produce *output contention*. Note that, since the workload in our tests is concentrated, our support may cause *output contention* because many messages are sent to the same destination, i.e. requests for chunk migration and coherence messages sent to overloaded processors. The Pull FM, to reduce the effects of output contention, moves data lazily, so that receivers pull data into local memories only when they become ready.

The results obtained by using Pull FM improve those shown in Fig. 9.(a), with reductions in the completion time which ranges between 0.2 and 0.3 sec. The improvements are greater (between 0.3 and 0.4 sec)



(a) (b)

Figure 9: Differences from the OCT for various μ as a function of the factor of imbalance F: (a) SUPPLE results, (b) CRAFT Fortran results.



(a) (b)

Figure 10: Case $F = 8$ and $\mu = 0.15$: (a) overheads due to *send*, *receive*, and *probe* MPI primitives, (b) chunks sent and received by each processor.

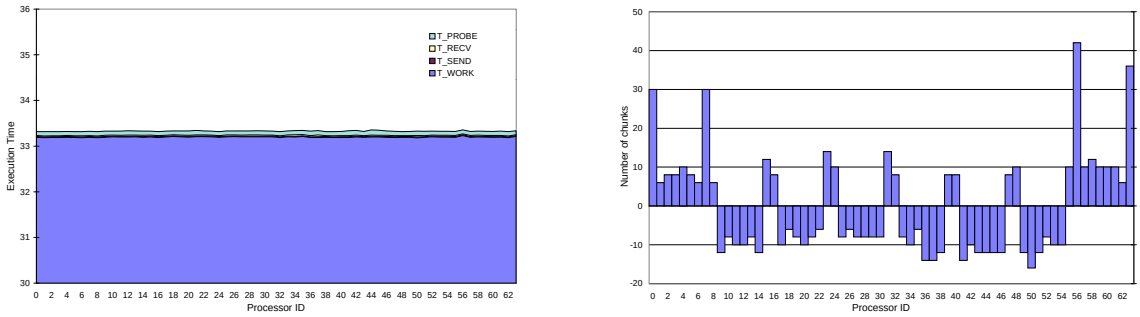
for the case $\mu = 0.3$, whose results appear to be the worst of those obtained by running SUPPLE on the CRI/EPCC MPI messaging layer.

Execution profile

We have profiled the execution of some tests carried out with different values of F and μ. Fig. 10.(a) shows, for all 64 processors, the time spent by each of them doing:

- useful work on local or remote chunks, but also some small service computations requested by our load balancing algorithm;
- sending, receiving, and checking for arrivals of messages.

Fig. 8 helps to understand this graph, which is relative to the case $F = 8$ and $\mu = 0.15$ *msec*, since it identifies the overloaded processors with respect to the input data set and the data layout. Recall that processors identified by 0, 1, 8, and 9 are overloaded, while processors 2, 10, 18, 16, and 17 are partially loaded. Going back to the graph of Fig. 10.(a), one can note a larger send/receive/probe overhead for the processors above. The reason for this overhead is the large amount of messages that they must dispatch to give away chunks, and to receive migration requests and coherence messages. Moreover, note that, due to these overheads, the processors that execute less useful work are the same processors that are overloaded at the beginning of computation. The graph in Fig. 10.(b) plots, for each processor, the expression



(a) (b)

Figure 11: Case $F = 1$ and $\mu = 0.15$: (a) overheads due to *send*, *receive*, and *probe* MPI primitives, (b) chunks sent and received by each processor.

$(Chunks_executed - Chunks_assigned)$. $Chunks_assigned$ is a constant: it is the number of chunks assigned to the local queue LQ of each processor during all the iterations of the reaction parallel loops. This number, considering that the simulation is iterated 10 times, is equal to 7560. $Chunks_executed$ is the interesting datum, and corresponds to the number of chunks (local + remote) that a processor actually executes. The expression is positive if a processor executes more chunks than those statically assigned: this is the case, as Fig. 10.(b) shows, of the underloaded processors, which receive further remote chunks. The same expression is negative if a processor executes less chunks than the initial ones: this is the case of the overloaded processors, which give away a lot of chunks to underloaded processors.

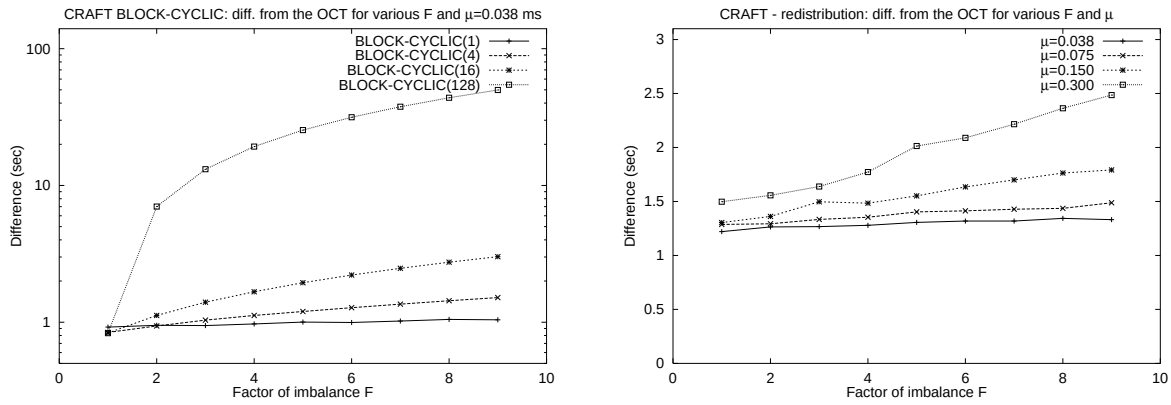
Fig. 11.(a) illustrates the moderate and uniform overheads introduced by SUPPLE for a balanced case characterized by $F = 1$ and $\mu = 15$ *msec*. Fig. 11.(b) shows, for the same case, the expression $(Chunks_executed - Chunks_assigned)$[1]. We can note a sort of symmetry between processors that export and import chunks. If we look at Fig. 8 (in this case, since $F = 1$, consider that the workload is uniformly distributed), we can discover that those processors that receive a few remote tasks are the same as those that are assigned blocks on the borders of the simulation grid, and processors on the corners are those that import the most chunks. This behavior is due to the first convection loop of our benchmark: this loop does not compute, in fact, the elements on the boundaries of the simulation grid, and thus introduces a low load imbalance which is detected and managed by SUPPLE during the reaction phase of the computation.

4.5 Comparison with CRAFT

We implemented the same benchmark with CRAFT Fortran, by feeding it with the same synthetic data sets. We tested several different data layouts. The best results for unbalanced problem instances were obtained by using a pure CYCLIC distribution. 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, due to this feature of our benchmark, it is more important to try to balance the workload through a pure CYCLIC distribution than to increase locality exploitation with a BLOCK distribution.

Fig. 9.(b) shows the results attained by CRAFT Fortran by adopting a CYCLIC 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 are worse for larger μ 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 μ equal to 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 μ, the workload

[1] Note that the scale of this graph is very different from the one of Fig. 10.(b).



(a) (b)

Figure 12: CRAFT Fortran differences from the OCT: (a) for different data layouts with $\mu = 0.038$ $msec$, (b) for different μ with array redistribution.

differences between overloaded and underloaded processors increase.

We also conducted other tests with CRAFT Fortran to evaluate combinations of BLOCK and CYCLIC distributions. By adopting a pure BLOCK distribution for the completely balanced case ($F = 1$) we obtained, as expected, the best CRAFT results. Also in this case, however, the SUPPLE results are better than the CRAFT ones. We believe that this is due to our very efficient implementation of stencil communications, the absence of barrier synchronizations, the small overheads introduced by our load balancing technique, and some migrations of chunks toward the processors holding the borders of the simulation grid (see Fig. 11.(b)). Some of the results obtained by adopting a combination of BLOCK and CYCLIC distributions are shown in Fig. 12.(a). All the curves, which refer to the case $\mu = 0.038$ $msec$, plots the differences from the optimum time for different values of F. The scale of the ordinate axis is logarithmic due to the very large differences from the optimal times when adopting a BLOCK data distribution for unbalanced cases.

Finally, we tested the redistribution of the simulation grid between consecutive parallel loop executions. CRAFT Fortran does not support run-time redistribution directives, so we used two different arrays, the former distributed BLOCK to use in the first convection loop, and the latter distributed CYCLIC to use in the second reaction phase. We re-assign one array to the other before the execution of each parallel loop. The results, which are shown in Fig. 12.(b) are worse than those obtained by using a pure CYCLIC distribution for both loops. The reason for this worse performance depends on the huge volume of data moved simultaneously between non adjacent processors which may cause network congestion problems. With regard to the amounts of data transferred during the whole simulation, Table 3 compares data volumes transmitted over the interconnection network by SUPPLE and CRAFT implementations of the benchmark. The data about SUPPLE refer to the case $\mu = 0.15$ $msec$: the small value is relative to $F = 1$, while the large one to $F = 9$. Note that for CRAFT CYCLIC, most of the communications derive from the implementation of the first convection loop, while for CRAFT redistribution it derives from the assignment statements used to redistribute the arrays. The values reported in the table refer to all the communications that derive from the execution of 10 time steps of both the convection and reaction phases. Data volumes moved in the case of a CYCLIC data layout are larger than the ones for REDISTRIBUTION. However, while in the CYCLIC implementation messages are exchanged only among nearest neighbor processors, in REDISTRIBUTION each processor must communicate with all the others to carry out array redistribution. Moreover the CRAFT Fortran compiler does not support array redistribution directives and thus, the communication schedules used to assign a cyclically distributed array to a block distributed one, and vice-versa, may not be fully optimized. The performance results reported in Fig. 12.(b) might be different if optimized array redistribution [21, 22] were supported.

Table 3: Data volumes transferred over the interconnection network

IMPLEMENTATION	DATA VOLUMES
SUPPLE	from 2.5 MB to 19.4 MB
CRAFT (BLOCK distrib.)	2.3 MB
CRAFT (CYCLIC distrib.)	320 MB
CRAFT (redistribution)	159.8 MB

5 Related work

The parallel loop scheduling problem has been investigated in depth by researchers working on shared–memory multiprocessors. Most proposals address the efficient implementation of loops by defining dynamic *Self Scheduling* policies which reduce synchronizations among processors by enlarging parallel task granularity. The main goal of these works is to determine the optimal size for the chunks fetched by each processor at each access to a shared queue which stores iteration indexes. Clearly, the larger the chunk size is, the lower the contention overheads for accessing the shared queue, and the higher the probability of introducing load imbalances. Polychronopoulos and Kuck have proposed *Guided Self Scheduling*, according to which $\frac{u}{P}$ iterations, where u is the number of remaining unscheduled iterations and P is the number of processors involved, are fetched at each time by an idle processor [23]. *Trapezoid Self Scheduling* [24] has been proposed by Tzen and Ni to reduce the number of synchronizations by linearly decreasing the chunk size. Hummel, Schonberg and Flynn have presented *Factoring* [16], the policy adopted also in SUPPLE to implement the *task selection* strategy of our load balancer. As mentioned in Section 3.2, *Factoring* requires that P consecutive chunks of size k, where $k \leq \frac{u}{2 \cdot P}$, are inserted into the shared queue when it becomes empty. The introduction of large caches on shared–memory multiprocessors makes the schema above unsuitable because they do not guarantee the exploitation of locality. Markatos and LeBlanc [25] have investigated locality and data reuse to obtain scalable and efficient implementations of non–uniform parallel loops. They have explored a scheduling strategy, based on a static partitioning of iterations, which initially assigns iterations for *affinity* with previously assigned ones. Affinity regards the presence of accessed data in the processor caches. The dynamic part of the technique, which performs run-time load balancing, is postponed until a load imbalance occurs. Liu and Saletore have worked on Self Scheduling techniques for distributed–memory machines [26]. They have designed hierarchical and distributed implementations of a shared queue manager, and have investigated partial data replication as a way to solve the data locality problem.

Willebeek-LeMair and Reeves [15] have presented several general load balancing strategies for multicomputers. They have introduced the framework that is used in Section 3.2 to characterize our load balancer. Another interesting work that presents general load balancing techniques is the one by Kumar et al. [14]. They introduce a technique that adopts a *global round-robin* policy to select the processing node to which a further task must be requested. The technique does not assume any knowledge of the load, and thus it may not be very accurate in scheduling decisions. On the other hand, the technique does not waste any time evaluating the best load balancing choices. Kumar et al. have shown that the technique is actually very scalable, provided that an efficient contention-free implementation of the *global round-robin* is adopted. We agree with their conclusions and we used a similar technique to implement our *Task Migration* strategy. However, our strategy is local rather than global since whenever a processor needs further work, it sends requests to some of its partners chosen on a simple inexpensive round-robin basis.

A lot of research has been carried out on run-time supports and compilation methods for distributed–memory multiprocessors. The Fortran D project at Rice and Syracuse universities [3], the Vienna Fortran project at the University of Vienna [2], the Fx Fortran compiler developed at Carnegie Mellon [27], and the CHAOS project at the University of Maryland [7], are only some of the most important projects addressing these topics. A number of compile-time optimizations have been designed to increase the performances obtained on distributed–memory multiprocessors [2, 3, 9]. The solution to problems whose irregularities prevent compile-time optimizations, is generally addressed by exploiting *inspector-executor* codes which collect information at run-time, and then use this information to optimize the successive computations [28]. The CHAOS/PARTI library [7] developed by the group headed by Joel Saltz at the University of Maryland,

adopted in the Vienna Fortran Compilation System and in other HPF-like compilers, is the most notable example of this approach. It provides support for irregular array distribution, loop scheduling based upon the *almost owner computes* rule, run-time data redistribution, data accessed through indirection arrays, thus offering general support for irregular/adaptive codes whose behavior cannot be anticipated until run-time. Non–uniform parallel loops can be implemented with the CHAOS library by collecting information about iteration execution time at run-time, and by consequently building an irregular data layout resulting in balanced processor workloads [10, 8]. Data partitioning and redistribution have to be repeated several times if the computational costs associated with the data items involved vary as execution progresses.

6 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 general BLOCK distributions of the involved arrays 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* 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. In most of the experiments we obtained completion times which differ by less than 1 *sec* from the optimal completion times. 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 times. 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. This is a valuable result since, due to the regularity of the shape of the loaded region in our synthetic input data set, the CYCLIC static distribution employed works well enough to reduce the, otherwise very large, load imbalance. We believe that SUPPLE will work much better than static HPF–like solutions for more irregular workload distributions.

Although our results are very satisfactory, SUPPLE can be improved by reducing the communication overheads that are incurred by processors and that, unlike latencies, cannot be hidden by overlapping computations. To this end, global address–based communications can be exploited to avoid the buffering costs of message–passing implementations. The machine we used for the tests, the Cray T3D, already provides dedicated hardware for remote read and write operations. The new Cray T3E uses enhanced virtual shared–memory mechanisms to make more usable these shared memory mechanisms, and also to implement more efficiently message–passing primitives [29]. Note, however, that remote–memory operations can be exploited to transfer data when the sender (receiver) knows the address to which it has to write (read). In our support, this is the case of communications which migrate data tiles toward underloaded processors, and receive back the updates. In fact, since a data tile is migrated to grant a request message, this message

may include a pointer to the location into which the data must be copied. Similarly, when an overloaded processor writes a data tile remotely, it may also communicate the memory location where the updates must be copied back. Note, however, that small messages are still needed to prefetch work and to communicate termination. The new Cray T3E also supplies efficient primitive mechanisms for small message queues. Mixed interrupt and polling policies can be used by allowing interrupts to arise only when polling begins not working properly. We hope that with these enhancements, SUPPLE overheads will be reduced further, while its generality and portability over different parallel machines will be maintained. Recent large–scale distributed–memory machines are in fact incorporating increasingly sophisticated architectural support to efficiently provide a global address space [30]. Following this architectural trend, one–side communications have been proposed as an important extension for the MPI standard as well [31].

References

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

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

[3] S. Hiranandani, K. Kennedy, and C. Tseng, "Compiling Fortran D for MIMD Distributed-Memory Machines," *Comm. of the ACM*, vol. 35, no. 8, pp. 67–80, Aug. 1992.

[4] J.M. Sipelstein and E. Blelloch, "Collection-Oriented Languages," *Proc. of the IEEE*, vol. 79, no. 4, pp. 504–523, April 1991.

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

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

[7] J. Saltz et al., "Runtime and Language Support for Compiling Adaptive Irregular Programs on Distributed Memory Machines," *Software Practice and Experience*, vol. 25, no. 6, pp. 597–621, June 1995.

[8] 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*, vol. 6, no. 8, pp. 815–831, Aug. 1995.

[9] S. Hiranandani, K. Kennedy, and C. Tseng, "Evaluating Compiler Optimizations for Fortran D," *Journal of Parallel and Distributed Computing*, vol. 21, no. 1, pp. 27–45, April 1994.

[10] 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.

[11] 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*, vol. 36, no. 7, pp. 845–858, July 1987.

[12] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of block algorithms," in *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991, pp. 63–74.

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

[14] V. Kumar, A.Y. Grama, and N. Rao Vempaty, "Scalable Load Balancing Techniques for Parallel Computers," *J. of Parallel and Distr. Comp.*, vol. 22, pp. 60–79, 1994.

[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*, vol. 4, no. 9, pp. 979–993, Sept. 1993.

[16] S.F. Hummel, E. Schonberg, and L.E. Flynn, "Factoring: A Method for Scheduling Parallel Loops," *Comm. of the ACM*, vol. 35, no. 8, pp. 90–101, Aug. 1992.

[17] V. Karamcheti and A. A. Chien, "A comparison of architectural support for messaging on the TMC CM-5 and Cray T3D," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, New York, June 22–24 1995, pp. 298–307, ACM Press.

[18] 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*, 1996, pp. 117–124.

[19] O. Maquelin, G. Gao, H. Hum, K. Theobald, and X. Tian, "Polling watchdog : Combining polling and interrupts for efficient message handling," in *Proceedings of the 23rd Annual International Symposium on Computer Architecure*, New York, May 22–24 1996, pp. 179–190, ACM Press.

[20] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, Gold Coast, Australia, May 19–21, 1992, ACM SIGARCH and IEEE Computer Society TCCA, pp. 256–266, *Computer Architecture News,* 20(2), May 1992.

[21] S. Ramaswamy and P. Banerjee, "Automatic generation of efficient array redistribution routines for distributed memory multicomputers," in *Proceedings of Frontiers '95: The Fifth Symposium on the Frontiers of Massively Parallel Computation*, Feb. 1995, pp. 342–349.

[22] K. Kennedy, N. Nedeljković, and A. Sethi, "Efficient address generation for block-cyclic distributions," in *Proc. of the 1995 ACM Int. Conf. on Supercomputing*, July 1995, pp. 180–184.

[23] C. Polychronopoulos and D.J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Trans. on Computers*, vol. 36, no. 12, Dec. 1987.

[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*, 1991, pp. 247–250.

[25] E.P. Markatos and T.J. LeBlanc, "Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors," *IEEE Trans. on Parallel and Distr. Systems*, vol. 5, no. 4, pp. 379–400, April 1994.

[26] J. Liu and V. A. Saletore, "Self-Scheduling on Distributed-Memory Machines," in *Proc. of Supercomputing '93*, 1993, pp. 814–823.

[27] T. Gross, D. O'Hallaron, and J. Subhlok, "Task parallelism in a high performance fortran framework," *IEEE Parallel and Distributed Technology*, vol. 2, no. 2, pp. 16–26, 1994.

[28] 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.*, vol. 22, no. 3, pp. 462–479, 1994.

[29] S. L. Scott, "Synchronization and Communication in the T3E Multiprocessor," in *Proc. of the Seventh Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Oct. 1996.

[30] A. Krishnamurthy, K. E. Schauser, C. J. Scheiman, R. Y. Wang, D. E. Culler, and K. Yelick, "Evaluation of Architectural Support for Global Address-Based Communication in Large-Scale Parallel Machines," in *Proc. of the Seventh Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Oct. 1996.

[31] Message–Passing Interface Forum, *MPI–2: Extensions to the Message–Passing Interface*, Nov. 1996, Draft.