Scheduling Data–Parallel Computations on Heterogeneous and Time–Shared Environments

Salvatore Orlando[1] and Raffaele Perego[2]

[1] Dip. di Matematica Appl. ed Informatica, Università Ca' Foscari di Venezia, Italy
[2] Istituto CNUCE, Consiglio Nazionale delle Ricerche (CNR), Pisa, Italy

Abstract. This paper addresses the problem of load balancing data–parallel computations on heterogeneous and time-shared parallel computing environments, where load imbalance may be introduced by the different capacities of processors populating a computer, or by the sharing of the same computational resources among several users. To solve this problem we propose a run–time support for parallel loops based upon a hybrid (static + dynamic) scheduling strategy. The results obtained on many experiments conducted on SGI/Cray T3E and IBM SP2 systems show that our hybrid scheduling strategy, which first allocates the same amount of work to each processor, and then dynamically balances the load, yields very good performances. The main features of our novel technique are the absence of centralization and synchronization points, the prefetching of work toward slower processors, and the overlap of communication latencies with useful computation.

1 Introduction

It is widely held that distributed and parallel computing disciplines are converging. In the 1980s the prevailing view was that parallel computers were suited for parallel applications composed of many fine–grain tasks cooperating in a tightly coupled fashion, while distributed computers were mainly used for loosely–coupled applications composed of few tasks with very coarse grains. Also the programming languages and tools used in the two fields were truly different.

As often happens, substantial technology improvements have forced this view to change. Advances in network technology have strongly increased the network bandwidth of state–of–the–art distributed systems. A gap still exists with respect to communication latencies on the two platforms, but this difference too is now much less dramatic. Furthermore, most Massively Parallel Systems (MPSs) are now built around the same off–the–shelf superscalar processors that are used in high performance workstations, so that fine–grain parallelism is now exploited intra–processor rather than inter–processor.

The convergence of parallel and distributed computing disciplines is also more evident if we consider the programming models and environments which dominate current parallel programming: MPI, PVM and High Performance Fortran (HPF) are now available for both MPSs and NOWs.

The trend we have briefly outlined has a direct impact on the research carried out on the two converging disciplines. For example, parallel computing research has to deal with problems introduced by heterogeneity in parallel systems. This is a typical feature of distributed systems, but nowadays heterogeneous NOWs are increasingly used as parallel computing resources [2], while MPSs may be populated with different off–the–shelf processors (e.g. a SGI/Cray T3E system at the time of this writing may be populated with 300 or 450 MHz DEC Alpha processors, while a Meiko CS-2 may be equipped with vector units on only some of its processing nodes). Furthermore, some MPSs, which were normally used as batch machines in *space sharing*, may now be concurrently used by different users as *time shared* multiprogrammed environments (e.g. an IBM SP2 system can be configured in a way that allows users to specify whether the processors must be acquired as shared or dedicated resources). This paper focuses on one of the main problems programmers thus have to deal with on both parallel and distributed systems: the problem of "system" load imbalance due to heterogeneity and time sharing of resources. Here we restrict our view to data–parallel computations expressed by means of parallel loops.

In previous works we considered imbalances introduced by non–uniform data–parallel computations to be run on homogeneous, distributed memory, MIMD parallel systems [12, 15, 13, 14]. We devised a novel compiling technique for parallel loops and a related run–time support (SUPPLE). This paper shows how the same support can also be utilized to implement loops in all the cases where load imbalance is not a characteristic of the user code, but is caused by variations in capacities of processing node. SUPPLE is based upon a hybrid scheduling strategy, which dynamically adjusts the workloads in the presence of variations of processor capacities. The main features of SUPPLE are the efficient implementation of regular stencil communications, the run–time scheduling of chunks of iterations, and the exploitation of aggressive chunk prefetching to reduce waiting times by overlapping communication with useful computation.

We report exhaustive performance results of many experiments carried out on an SGI/Cray T3E and an IBM SP2 system. The synthetic benchmarks used for the experiments allowed us to model different situations by varying a few important parameters such as the computational grain and the capacity of each processor. The results obtained suggest that, in the absence of a priori knowledge about the relative capacities of the processors that will actually execute the program, the hybrid strategy adopted in SUPPLE yields very good performance.

The paper is organized as follows. Section 2 briefly surveys related work, while Section 3 presents the synthetic benchmarks and the machines used for the experiments. Section 4 describes our run–time support and its load balancing strategy. The experimental results are reported and discussed in depth in Section 5, and, finally, the conclusions are drawn in Section 6.

2 Related work

Since data–parallelism is widely considered the best suited paradigm for many important problems in various application fields, in the last few years high–level data–parallel languages such as HPF [3], Vienna Fortran [20] and Fortran D [4] have received considerable interest. While the static compilation model generally adopted for HPF works well for regular problems and homogeneous space–shared architectures, workload imbalance due to the irregularities of either the problem or the target machine may worsen the final performance.

Much research has been conducted in the field of run-time supports and compilation methods for irregular problems [18, 16, 10, 12, 15, 13, 14]. In our opinion, many of these techniques can be also adopted when load imbalance derives from the use of a time–shared or heterogeneous parallel system. Other techniques have been specifically devised to solve the problem of load imbalance when this is introduced into uniform data parallel computations by heterogeneous and/or time–shared environments. *Weighted Factoring* [6] has been conceived for heterogeneous distributed environments (NOWs), and is a variant of the well–known *factoring* self–scheduling scheme [7] designed for shared–memory environments. The main drawback of the proposal is still the presence of a centralized scheduler processor. Moreover, it considers the relative speeds of processors involved as an input of the load balancing technique, even though these speeds may change during the computation due to time–sharing of resources.
DAME [11] aims to extend the Single Program Multiple Data (SPMD) programming paradigm to better harness heterogeneous NOWs with time-varying workloads. A centralized master collects information at run-time on the load of the workstations involved, and explicit primitives are furnished to programmers in order to activate load checking and redistribute arrays in blocks whose sizes depend on the actual capacities of each workstation. A drawback of the proposal, besides the presence of a single point of centralization, is the introduction of a global synchronization with consequent data migration to change at run-time the shapes of sub-blocks owned by each processor. Moreover, the redistribution may change the adjacency relationships among those processors that might have to exchange borders to implement regular stencil data references.

3 Benchmarks

We adopted a very simple benchmark program that resembles a very common pattern of parallelism (e.g. solvers for differential equations, simulations of physical phenomena, and image processing applications). The pattern is *data–parallel* and "regular", and thus considered easy to implement on homogeneous and dedicated parallel systems.

In the benchmark a bidimensional array is recomputed iteratively on the basis of the old values of its elements, while array data referenced are modeled by a *five-point stencil*. The simple HPF code illustrating the benchmark is shown below.

```
      REAL A(N1,N2)
      REAL B(N1,N2)
!HPF$ TEMPLATE D(N1,N2)
!HPF$ ALIGN A(i,j), B(i,j) WITH D(i,j)
!HPF$ DISTRIBUTE D(BLOCK,BLOCK)
      .........
      DO k= 1,N_ITER
!HPF$   INDEPENDENT
        FORALL (i= 2:N1-1,  j= 2:N2-1)
          B(i,j) = Comp(A(i,j), A(i-1,j), A(i,j-1), A(i+1,j), A(i,j+1))
        END FORALL
        A = B
```

```
END DO
.........
```

Note the BLOCK distribution to exploit data locality by reducing off local-memory data references. The actual computation performed by the benchmark above is hidden by the function `Comp()`. We thus prepared several versions of the same benchmark where `Comp()` was replaced with a dummy computation characterized by known and fixed costs.

Moreover, since it is important to observe the performance of our loop support when we increase the number of processors, for each different grid `P` of processors we modified the dimensions of the data set to keep the size of the block of data allocated to each processor constant.

Finally, another feature that we changed during the tests is `N_ITER`, the number of iterations of the external sequential loop. This was done to simulate the behavior of real applications such as solvers of differential equations, which require the same parallel loops to be executed many times, and image filtering applications, which usually perform the update of the input image in just one step.

4 The SUPPLE approach

SUPPLE (SUPport for Parallel Loop Execution) is a run-time support for parallel loops [12, 13, 15, 14]. It is written in C with calls to the MPI library.

The parallel loop support The main innovative feature of SUPPLE [13] is its ability to allow data and computation to be dynamically migrated, without losing the ability to exploit all the static optimizations that can be adopted to efficiently implement stencil data references.

Stencil implementation is straightforward, due to the regularity of the *blocking* data layout adopted. For each array involved SUPPLE allocates to each processor enough memory to host the block partition, logically subdivided into an *inner* and a *perimeter* region, and a surrounding *ghost* region. The *ghost* region is used to buffer the parts of the *perimeter* regions of the adjacent partitions that are owned by neighboring processors, and are accessed through non local references. The *inner* region contains data elements that can be computed without fetching external data, while to compute data elements belonging to the *perimeter* region these external data have to be waited for. Loop iterations are statically assigned to each processor according to the *owner computes rule*, but, to overlap communications with useful computations, iterations are reordered [5]: the iterations that assign data items belonging to the *inner* region (which refer local data only) are scheduled between the asynchronous sending of the *perimeter* region to neighboring processors and the receiving of the corresponding data into the *ghost* region.

This static scheduling may be changed at run-time by migrating iterations, but, in order to avoid the introduction of irregularities in the implementation of stencil data reference, only iterations updating the *inner* region can be migrated.

The hybrid scheduling strategy In this paragraph we focus on the hybrid scheduling of iterations updating the *inner* block partition. We group these iterations into *chunks* of fixed size, by statically *tiling* the inner region. Tiling is a useful optimization which improves locality exploitation [9] and, in this case, is also exploited by our hybrid scheduling technique. In fact, in order to balance the workload, SUPPLE migrates chunks and associated data tiles instead of single iterations.

At the beginning, to reduce overheads, each processor statically executes its chunks, which are stored in a queue Q, hereafter *local* queue. Once a processor understands that its local queue is *becoming empty*, it autonomously decides to start the dynamic part of its scheduling policy. It tries to balance the processor workloads by asking overloaded partners for migrating both chunks and corresponding data tiles. Note that, due to stencil data references, to allow the remote execution of a chunk, the associated tile must be accompanied by a surrounding area, whose size depends on the specific stencil features. Migrated chunks and data tiles are stored by each receiving processor in a queue RQ, called *remote*. Since load balancing is started by underloaded processors, our technique can be classified as *receiver initiated* [8, 19]. In the following we detail our hybrid scheduling algorithm.

During the initial *static* phase, each processor only executes local chunks in Q and measures their computational cost. Note that, since the possible load imbalance may only derive from different "speeds" of the processors involved, chunks that will possibly be migrated and stored in RQ will be considered as having the same cost as the ones stored in Q. Thus, on the basis of the knowledge of the chunk costs, each processor estimates its *current load* by simply inspecting the size of its queues Q and RQ.

When the estimated local load becomes lower than a machine-dependent *Threshold*, each processor autonomously starts the *dynamic* part of the scheduling technique and starts asking other processors for remote chunks. Correspondingly, a processor p_j, which receives a migration request from a processor p_i, will grant the request by moving some of its workload to p_j only if its *current load* is higher than *Threshold*. To reduce the overheads which might derive from requests for remote chunks which cannot be served, each processor, when its *current load* becomes lower than *Threshold*, broadcasts a so-called *termination message*. Therefore the *round-robin* strategy used by underloaded processors to choose a partner to be asked for further work skips terminated processors.

Once an overloaded processor decides to grant a migration request, it must choose the most appropriate number of chunks to be migrated. To this end, SUPPLE uses a modified *Factoring* scheme [7], which is a *Self Scheduling* heuristics formerly proposed to address the efficient implementation of parallel loops on shared–memory multiprocessors, where a single queues of iterations is concurrently accessed. In our case, rather than a single queue, we have multiple shared queues, one for each processor, and the scheduling unit is the chunk instead of a single iteration. The modified heuristic 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.

Finally, the policies exploited by SUPPLE to manage data coherence and termination detection are also fully distributed and asynchronous. A *full/empty-like* technique [1] is used to asynchronously manage the coherence of migrated data tiles. When processor

p_i sends a chunk b to p_j, it sets a flag marking the data tiles associated with b as invalid. The next time p_i needs to access the same tiles, p_i checks the flag and, if the flag is still set, waits for the updated data tiles from node p_j. Such a fine control of data consistency does not entail waiting for coherence messages at the end of parallel loop execution: a new loop can be started and useful computations can be overlapped with the latencies of these communications. As far as termination detection is concerned, the role of a processor in the parallel loop execution finishes when it has already received a termination message from all the other processors, and both its queues Q and RQ are empty.

In summary, since SUPPLE only exploits local knowledge to make dynamic scheduling decisions, there is no need to synchronize the processors in order to exchange updated information about the global workload. Moreover, unlike other proposals [16, 6], its dynamic scheduling policy is fully distributed. Finally, unlike other proposed techniques [18, 17, 11], SUPPLE may also be employed for applications composed of a single parallel loop, such as filters for image processing. It does not exploit past knowledge about the workload distribution at previous loop iterations, since dynamic scheduling decisions are asynchronously taken concurrently with useful computation.

5 Experimental results

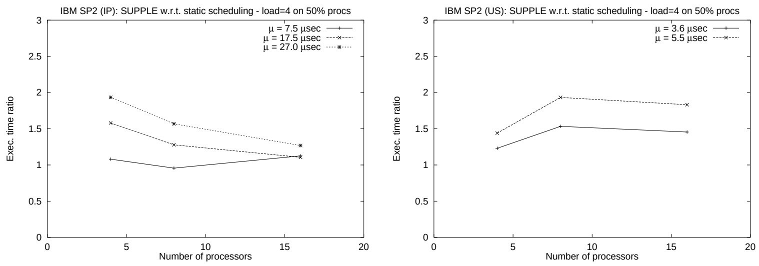
All experiments were conducted on an IBM SP2 system and an SGI/Cray T3E. Note that both machines might be heterogeneous, since both can be equipped with processors of different capacities. The T3E can in fact be populated with DEC Alpha processors of different generations[3], while IBM enables choices from three distinct types of nodes – high, wide or thin – where the differences are in the number of per–node processors, the type of processors, the clock rates, the type and size of caches, and the size of the main memory.

However, we used the SP2 (a 16 node system equipped with 66 MHz POWER 2 wide processors) as a *homogeneous time–shared environment.* To simulate load imbalance we simply waited for a period of time during which the parallel machine was being idle, and then launched some compute–bound processes on a subset of nodes to introduce an imbalance in the processor capacities. After the reaching of the desired system load imbalance, we finally ran our tests.
On the other hand, we used the T3E (a system composed of 64 300 MHz - DEC Alpha 21164 processors) as a space-shared heterogeneous system. Since all the nodes of our system are identical, we had to simulate the presence of processors of different speeds by introducing an extra cost in the computation performed by those processors considered "slow". Thus, if the granularity of `Comp()` is μ μsec (including the time to read and write the data) on the "fast" processors, and F is the *factor of slowdown* of the "slow" ones, the granularity of `Comp()` on the "slow" processors is $(F \cdot \mu)$ μsec.

To prove the effectiveness of SUPPLE, we compared each SUPPLE implementation of the benchmark with an equivalent static and optimized implementation, like the one exploited by a very efficient HPF compiler.

[3] The Pittsburgh Supercomputing Center has a T3E system with 512 application processors of which half runs at 300 MHz and half at 450 MHz.

In this section we present several curves, all plotting an *execution time ratio* (ETR), i.e. the ratio of the time taken by the static scheduling version of the benchmark *over* the time taken by the SUPPLE hybrid scheduling version. Hence, a ratio greater than one corresponds to an improvement in the total execution time obtained by adopting SUPPLE with respect to the static version. Each curve is relative to a distinct granularity μ of `Comp()`, and plots the ETRs as a function of the number of processors employed. The size of the data set has been modified according to the number of processors to keep the size of the sub-block statically assigned to each processor constant.



(a) (b)

Fig. 1. SP2 results: ETR results for experiments exploiting (a) the Ethernet, and (b) the high–performance switch

5.1 Time–shared environments

First of all, we show the results obtained on the SP2. Due to the difficulty in running probing experiments because of the unpredictability of the workloads, as well as for the exclusive use of some resources (in particular, resources as the SP2 high–performance switch and/or the processors themselves) by other users, the results in this case are not so exhaustive as those reported for the tests run on the T3E. As previously mentioned, on the SP2 we ran our experiments after the introduction of a synthetic load on a subset of the nodes used. In particular, we launched 4 compute–bound processes on 50% of the processing nodes employed, while the loads were close to zero on the rest of the nodes. This corresponds to "slow" processors characterized by a value of F (*factor of slowdown*) equal to 5. The two plots in Figure 1 show the ETRs obtained for various μ and numbers of processors. The size of the sub-block owned by each processor was kept constant and equal to 512×512, while `N_ITER` was equal to 5.
Figure 1.(a) shows the SP2 results obtained by using the Ethernet network and the IP protocol. Note that, even in this case as in the following ones, the performance improvement obtained with SUPPLE increases in proportion to the size of the grain μ. For $\mu = 27$ μsec, and 4 processors, we obtained the best result: SUPPLE implementation is about 100% faster than the static counterpart. For smaller values of μ, due to the overheads to migrate data, the load imbalance paid by the static version

of the benchmark is not large enough to justify the adoption of a dynamic scheduling technique.
Figure 1.(b) shows, on the other hand, the results obtained by running the parallel benchmarks in time–sharing on the SP2 by exploiting the US high–performance switch. Due to the better communication framework, in this case the ETR is favorable to SUPPLE even for smaller values of μ.

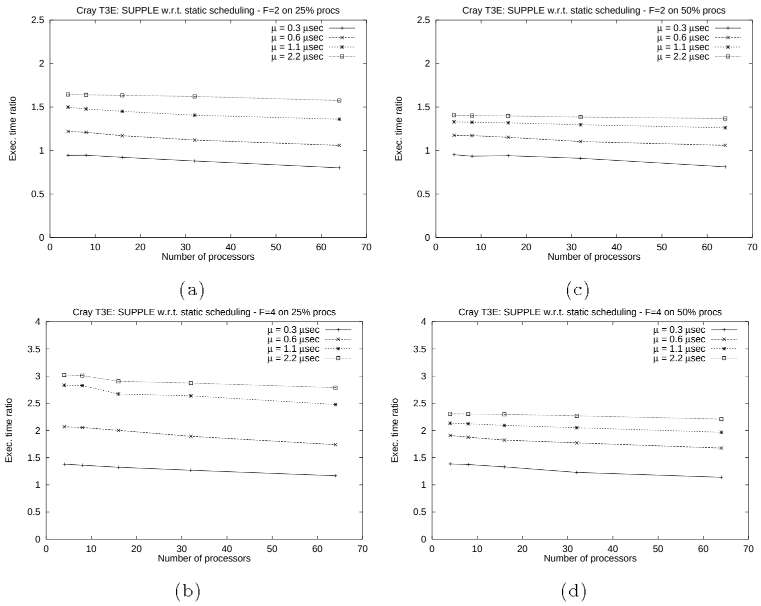


Fig. 2. T3E results: ETR results for various values of F and μ

5.2 Heterogeneous environments

All the experiments regarding heterogeneous environments were conducted on the T3E. Heterogeneity was simulated, so that we were able to perform a lot of tests, also experimenting different factors F of slowdown.

Iterative benchmark Figure 2 reports the results for the iterative benchmark, where the external sequential loop is repeated 20 times (`N_ITER = 20`). For all the tests we kept fixed the size of the sub–block assigned to each processor (512×512). Figures 2.(a) and 2.(b) are relative to an environment where only 25% of all processors are "slow". The factors F of slowdown of these processors are 2 and 4, respectively. Figures 2.(c) and 2.(d) refer to an environment where half of the processors employed are "slow'.
Each curve in all these figures plots the ETR for a benchmark characterized by a distinct μ. With respect to the SP2 tests, in this case we were able to reduce μ (μ now ranges between 0.3 and 2.2 μsec), always obtaining an encouraging performance

with our SUPPLE support. Note that such grains are actually very small: for μ equal to 0.3μsec, about 85% of the total time is spent on memory accesses (to compute addresses and access data element covering the five-point stencil), and only 15% on arithmetic operations. Looking at Figure 2, we can see that better results (SUPPLE execution times up to 3 times lower than those obtained with static scheduling) were obtained when the "slow" processors were only 25% of the total number. The reason for this behavior is clear: in this case, we have more "fast" processors to which the extra workload previously assigned to the "slow" processors can be dynamically distributed. The execution times obtained with the static implementation are, on the other hand, almost independent of the percentage of "slow" processors. In fact, even if only one of the processors was "slow", its execution time would dominate the overall completion time.

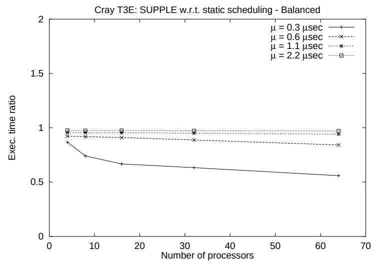
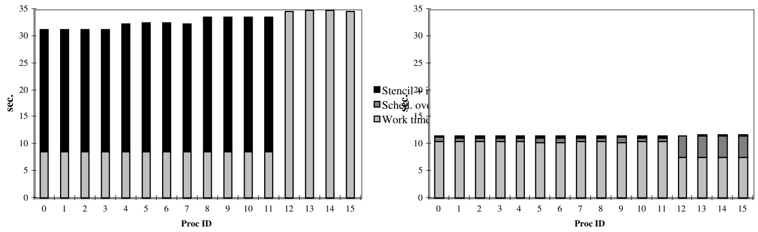


Fig. 3. T3E results: ETR results for a homogeneous (balanced) environment

Figure 3 shows the results obtained for a limit case, i.e. a homogeneous system on which we still try to use the SUPPLE version of the benchmark to evaluate the overhead introduced by our hybrid scheduling strategy w.r.t. the static implementation, which, in this case, is optimal. As we can see, the overhead was very small for larger values of μ, so that for $\mu = 2.2$ μsec the two execution times are almost comparable. For smaller μ, the static version of the benchmark becomes 60% faster than in SUPPLE. We verified that the overhead introduced by SUPPLE is due to some undesired migration of chunks, and to the dynamic scheduling technique which entails polling the network interface to check for incoming messages (even if these messages do not actually arrive). This overhead is almost constant for data sets of the same sizes subdivided into a given number of chunks, but its influence becomes larger for smaller granularities because of the shorter execution time and the limited possibility of hiding communication latencies with computations.

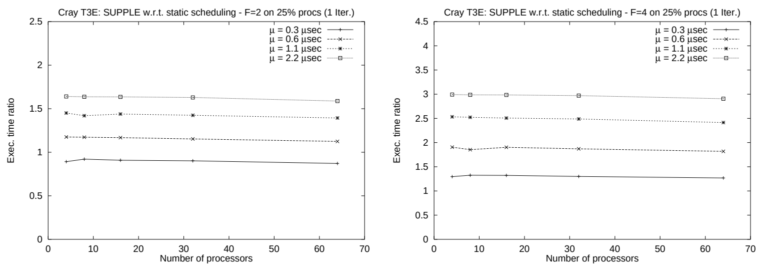
Finally, we instrumented the static and the SUPPLE versions of a specific benchmark to evaluate the ratio between the execution time spent on useful computations and on dynamic scheduling overheads. The features of the benchmark used in this case are the following: a 2048×2048 data set distributed over a grid of 4×4 T3E processors, and an external sequential loop iterated for 20 times. The simulated unbalanced environment was characterized by 25% of "slow" processors, with a slowdown factor of 4. Figure 4.(a) show the results obtained by running the static implementation of this



(a) (b)

Fig. 4. Work time and overheads for a static (a) and a SUPPLE-based (b) version of the benchmark

benchmark. It is worth noting the work time on the "slow" processors, which is 4 times the work time on the "fast" ones. The black portions of the bars show idle times on "fast" processors while waiting for border data. These idle times are thus due to communications implementing stencil data references, where corresponding send/receive on fast/slow processors are not synchronized due to their different capacities.
Figure 4.(b), on the other hand, shows the SUPPLE results on the same benchmark. Note the redistribution of workloads from "slow" to "fast" processors. Thanks to dynamic load balancing, idle times disappeared, but we have larger scheduling overheads due to the communications used to dynamically migrate chunks. These overheads are clearly larger on "slow" processors, which spend a substantial part of their execution time on giving away work and on receiving the results of migrated iterations.



(a) (b)

Fig. 5. T3E results: ETR results obtained running a *single iteration* benchmark for various values of F and μ

Single iteration benchmark As explained above, one of the advantages of SUPPLE is that it can also be used for balancing parallel loops that have only to be executed once. In this case some overheads cannot be overlapped, and at the end of loop execution "slow" processors have to wait for the results of iterations executed remotely.

Figure 5 shows the encouraging ETRs obtained by our SUPPLE implementation w.r.t. the static one. Note that all the results plotted in the figure refer to unbalanced environments where only 25% of the processors are "slow". Moreover, due to the larger data sets used for these tests, the ETRs are in some cases even more favorable for SUPPLE than the ones for iterative benchmarks.

6 Conclusions

We have discussed the implementation on heterogeneous and/or timed–shared parallel machines of regular and uniform parallel loop kernels, with statically predictable stencil data references. We have considered the problem of load imbalance introduced by the different capacities of the processors, due to their diversity and/or time–sharing. We have assumed that no information about the capacities of the processors involved is available until run-time, and that, in time–shared environments, these capacities may change during run–time.

To implement the kernel benchmark we employed SUPPLE, a run-time support that we had previously introduced for loops where the costs of iterations are non–uniform and unpredictable due to problem irregularities. Here we have shown that the same support can also be used in the presence of "system" load imbalance.

The tests were conducted on an SGI/Cray T3E and an IBM SP2. We compared the SUPPLE results with a static implementation of the benchmark, where data and computations are evenly distributed to the various processing nodes. The SP2, a parallel system that can be used as a time–shared NOWs, was loaded with artificial compute-bound processes before running the tests. On the other hand, we needed to simulate a heterogeneous SGI/Cray T3E, i.e. a machine whose nodes may be equipped with different off–the–shelf processors and/or memory organization.

The performance results were very encouraging. On the SP2, where half of the processors were loaded with 4 compute-bound processes, the SUPPLE version of the benchmark resulted at most 100% faster than the static one. On the T3E, depending on the amount of "slow" processors, on the number of processors employed and the granularity of loop iterations, the SUPPLE version reached percentages of performance improvement ranging between 20% and 270%.

We believe that hybrid strategies like the one adopted by SUPPLE can be profitably exploited in many cases where locality exploitation and load balance must be solved at the same time. The compilation techniques of data–parallel languages should be extended to exploit SUPPLE–like run–time supports, above all when the target machine is heterogeneous and time–shared, as are the very popular NOWs.

Acknowledgments We acknowledge the CINECA of Bologna for the use of their computational facilities.

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. A. Anurag, G. Edjlali, and J. Saltz. The Utility of Exploiting Idle Workstations for Parallel Computation. In *Proc. of the 1997 ACM SIGMETRICS*, July 1997.

3. High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993. Version 1.0.
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. Flynn Hummel, J. Schmidt, R. N. Uma, and J. Wein. Load-Sharing in Heterogeneous Systems via Weighted Factoring. In *Proc. of the 8th SPAA*, July 1997.
7. 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.
8. 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.
9. 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.
10. J. Liu and V. A. Saletore. Self-Scheduling on Distributed-Memory Machines. In *Proc. of Supercomputing '93*, pages 814–823, 1993.
11. M. Colajanni M. Cermele and G. Necci. Dynamic Load Balancing of Distributed SPMD Computations with Explicit Message-Passing. In *Proc. of the IEEE Workshop on Heterogeneous Computing*, pages 2–16, 1997.
12. 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.
13. S. Orlando and R. Perego. SUPPLE: an Efficient Run–Time Support for Non–Uniform Parallel Loops. Technical Report TR-17/96, Dipartimento di Mat. Appl. ed Informatica, Università di Venezia, Dec. 1996.
14. S. Orlando and R. Perego. A Comparison of Implementation Strategies for Non–Uniform Data–Parallel Computations. Technical Report TR-9/97, Dipartimento di Mat. Appl. ed Informatica, Università di Venezia, April 1997. Under revision for publication on the J. of Parallel and Distr. Comp.
15. S. Orlando and R. Perego. A Support for Non–Uniform Parallel Loops and its Application to a Flame Simulation Code. In *Proc. of the 4^{th} Int. Symposium, IRREGULAR '97*, pages 186–197, Paderborn, Germany, June 1997. LNCS 1253, Spinger-Verlag.
16. O. Plata and F. F. Rivera. Combining static and dynamic scheduling on distributed–memory multiprocessors. In *Proc. of the 1994 ACM Int. Conf. on Supercomputing*, pages 186–195, 1994.
17. 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.
18. J. Saltz et al. Runtime and Language Support for Compiling Adaptive Irregular Programs on Distributed Memory Machines. *Software Practice and Experience*, 25(6):597–621, June 1995.
19. 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.
20. H.S. Zima and B.M. Chapman. Compiling for Distributed-Memory Systems. *Proc. of the IEEE*, pages 264–287, Feb. 1993.

This article was processed using the LaTeX macro package with LLNCS style