An Efficient Template for the Highly Parallel Implementation of Non–Uniform Loops

Salvatore Orlando
Dip. di Matematica Appl. e Informatica
Università Ca' Foscari di Venezia
E-mail: orlando@unive.it

Raffaele Perego
Istituto CNUCE
Consiglio Nazionale delle Ricerche (CNR)
E-mail: r.perego@cnuce.cnr.it

Abstract

In this paper we present an efficient template for the implementation on distributed-memory multiprocessors of non–uniform parallel loops. The technique relies upon a static blocking distribution of the data sets and a hybrid scheduling policy. The template initially adopts a static technique to distribute the loop iterations among the processing nodes. As soon as a workload imbalance is detected, it exploits a dynamic receiver initiate technique to move work towards unloaded processors. Prefetching is used to reduce overheads of the technique due to the communications needed to monitor the load, move iterations, and restore the consistency of migrated data. The possibility of exploiting data locality deriving from the adoption of a blocking data distribution is one of the main goals of the proposed technique. A cyclic distribution which causes the loss of data locality should be instead used with an HPF-like language to achieve acceptable performances. Experiments have been conducted on a 64-node Cray T3D, and the performance of the proposed template has been compared with the one obtained by using the CRAFT-Fortran language (an HPF-like language).

1 Introduction

Parallel loops operating on arrays are the most common way to express data parallelism in many scientific compute-intensive applications. Therefore, a lot of research on parallel languages is focused on finding good run-time supports for the execution of loops on shared-memory symmetric multiprocessors and, recently, on distributed-memory multiprocessors.

One of the main issues in implementing parallel loops is the allocation of data with respect to the schedule of iteration. In UMA shared-memory multiprocessors, where, in principle, shared data are at the same distance from any processor, the usual implementation schemes rely on completely dynamic self-scheduling techniques, based on the existence of a global queue of iterations [8, 9, 3]. The introduction of caches makes this scheme unsuitable even for UMA multiprocessors, since it does not guarantee the exploitation of locality [6]. In distributed-memory multiprocessors, data distribution and iteration scheduling are also more strictly related. Data parallel languages such as HPF rely on user-provided directives to derive data distribution. Scheduling is usually static, and depends on the data distribution. The *owner computes rule* [10] is normally used to determine static schedules from data distribution. Programmers can specify alignments of arrays to each other, and distribution of arrays to processors according to some standard policies, i.e. *cyclic* and *blocking*. The main purpose of the *blocking* distribution directive is to give the compiler more information which can be used to produce a code which exploits locality, i.e. a code which is characterized by limited communication overheads. Conversely, *cyclic* distribution is good for problems where locality is less important than load balance issues. This occurs when the work required for each section of the data array varies significantly.

The choice of the best data distribution is currently up to programmers. It is known that, depending on some features of parallel loops (mainly, array references), the problem of finding an optimal data layout may be in NP [4]. On the other hand, it is likely that, for the cases where data references are either constant or are simple linear combinations of indexes, future compilers will be sophisticated enough to automatically align and distribute arrays, and thus obtain acceptable performances. These smart compilers will not be able, however, to solve cases in which, by using simple *blocking* or *cyclic* data distribution schemes, only one of the two contrasting goals is achieved, i.e. either locality or load balancing. Moreover, no automatic and semi-automatic *static tools* will be able to devise optimal data distributions when array references and computational loads of each iteration change at run-time in an unpredictable way.

This paper presents a technique to implement an efficient template explicitly designed to smooth any *unbalanced processor workload* deriving from loops whose independent iterations are characterized by highly varying execution times (*non–uniform* parallel loops) [7]. Two slightly different version of the template are presented. The first version is optimized for *iterated parallel loops*, i.e. parallel loops nested in a sequential one, for which it is necessary to mantain the coherence of the data among successive iterations of the outer sequential loop. Both data and iteration indexes are in this case moved across the interconnection network. The second version is instead designed for parallel loops which are not nested or which do not modify the accessed data. In this case static partial replication of the data set is exploited to minimize the run–time overhead of the technique. Both versions of the template rely upon a *blocking* data distribution to exploit locality, and combine static and dynamic policies to obtain well-balanced iteration schedules.

The paper describes the implementation template, and presents the experiments conducted on a 64 node Cray T3D. To compare our technique with an HPF-style implementation, we used the Cray CRAFT-Fortran, whose language directives and compilers have been adapted in part from Rice University's Fortran-D project [2] and Vienna Fortran [10].

2 The technique

Our technique assumes that accessed arrays, which may be *aligned* to an abstract topology, are partitioned into P contiguous blocks, and each block b_i is statically assigned to one of the P processor, say p_i. We call p_i *owner* of block b_i. Iterations operating on each partition are subdivided into *chunks* of fixed size and assigned to the various processors according to the *owner computes rule.* Each processor has a local queue Q to store its own chunks. At the beginning, and until a load imbalance has been detected, chunks are statically scheduled by fetching them from Q. The dynamic part of the scheduling policy can be classified as a *receiver initiated technique* [5] and starts when a processor understands that queue Q is *becoming empty.* Another queue QR, initially empty, is used at this point to store the chunks received from more loaded partners.

Figure 1 sketches the SPMD implementation of an iterated parallel loop. In particular, the figure shows the pseudo-code executed by a generic processor p_i, $i \in \{0, \ldots, P-1\}$. As can be seen, two distinct arrays `partners[]` and `is_partner_of[]` characterize the code of each processor p_i. The former array contains a subset of so-called *partner* processors, $\{p_{i_1}, \ldots, p_{i_m}\}$, $m < P$. These are the processors to which p_i will, if necessary, ask for further work if a load imbalance occurs. For each p_i, our technique requires the *static* assignment of a distinct set of m partners, with the property that each p_i is assigned as a partner to a set of exactly m other processors. This last set of m processors is stored in the latter array `is_partner_of[]`, and is used by p_i to find out from which processor it might be asked to migrate work; m is defined as the *partnership degree* of our technique.

```
Parallel_Loop (queue Q, process partners[ ], is_partner_of[ ])
QR: queue; T: chunk;
  begin
    Initialize_Data_Structures ();
    while (not Terminated (Q, QR, partners)) do
      if (My_Load(Q) < Threshold) then
        Prefetch_Chunks (partners);
        if (* My_Load(Q) < Threshold for the first time *) then
          Comm_unloaded (is_partners_of);
        endif
      endif
      T = Extract(Q);
      if (Empty (T)) then
        T = Extract (QR);
        Execute (T);
        Send_Results (T);
      else
        Execute (T);
      endif
      Request_Handler ();
    endwhile
  end
```

Figure 1: Pseudo-code of the scheduling algorithm.

In the version of the template which exploits data replication, each block b_i, owned by processor p_i, is statically copied also on the local memories of the m processors whose names are stored in array `is_partner_of[]`. Thus, when a partner processor p_j asks p_i for a chunk to be migrated, only chunk indexes have to be communicated. Simmetrically, when p_i requires a chunk to one of its partner processors (array `partners[]`), it receives only the indexes of the migrated chunk since it already holds a copy of the data needed to execute it. Data replication clearly reduces the amount of data that have to be transferred but introduces high overheads if also the coherence of replicated data must be maintained. Thus we chose to exploit this technique only in the cases of single parallel loops or when the replicated data set is read–only. For generic iterated parallel loops the version of the template which transfer at run–time also the data should result in better performances.

The main features of our scheduling techniques are:

- A *processor load evaluation* based on local information only. Since the first part of our technique is static, the average cost of local chunks can be determined by a very simple code instrumentation, which accumulates the time spent by chunk executions. We therefore compute the *local load* of a processor (function *My_load()*) as the number of chunks currently stored in Q times the average execution cost of the chunks already executed.
- An effective prefetching policy which reduces overheads of chunk migration by overlapping computation and communication. It is driven by a *Threshold* parameter: a processor asks a partner (*Prefetch_Chunk(partners)*) for a new chunk only if its local load is less than *Threshold*. Similarly, the partner process serves the requests only if its own local load is greater than *Threshold*.
- A simple local *round-robin* policy to select the next partner to be asked for a further chunk. Note that the next partner is selected from those which still have not communicated that their load has become smaller than *Threshold*. Each processor communicates this event to all the processors from which it may receive a chunk request by means of the subroutine *Comm_unloaded(is_partner_of)*. Note that these communications (*termination messages*) are also used to implement a distributed *termination protocol*. The function *Terminated(Q, QR, partners)* checks termination: p_i terminates the parallel loop as soon as its queues Q and QR are empty and it has received a termination message from all the members of `partner[]`.
- An asynchronous *coherence protocol* to migrate updated data from the processor that actually performed the chunk to the processor that is the *owner* of the corresponding data partition. The updated data are transmitted by the subroutine *Send_Results (T)*. When the parallel loop is iterated, a *full/empty-like* technique [1] is used to avoid processing invalid data. When processor p_i sends a chunk c_j to p_{i_k}, it sets a flag indicating that the data to be modified by c_j are no longer valid. The next time p_i needs to access the same data, e.g. during the next iteration of the outer sequential loop, p_i checks the flag and, if the flag is still set, waits for the updated data from node p_{i_k}. Such control of data coherence does not require barrier synchronizations at the end of each execution of the parallel loop.

Request_Handler(Q, QR), whose code, for the sake of simplicity, is not shown in detail, performs many of the above tasks. This subroutine handles all the messages that can arrive at a processor, i.e. requests for more work from an unloaded processor (this may require a chunk to be extracted from Q and sent to the unloaded processor), coherence messages transporting updated data, further chunks sent by partner processors (these chunks are inserted in QR), and service messages which signal that the local queue of a partner process is becoming empty (termination messages).

3 Results and Future Work

The benchmark that we have used to validate the idea is a parallel loop which operates on a large bi-dimensional array of floating point numbers. The computation performed by the parallel loop is simply a function that is applied to all the elements in the array to produce the new ones. The execution time of each function instance is directly proportional to the value of the processed array element. If μ is the average execution cost of parallel loop iterations, and D is the number of iterations, $T = D \cdot \mu$ is the total workload of the parallel loop. If the parallel loop is iterated K times, the total workload is $K \cdot T$.
The model we used to generate the various arrays `A[]`, each corresponding to a problem characterized by a given unbalanced workload, is based on two parameters, d and t. The parameter d, $0 < d < 1$, determines the fraction of the whole array on which the per-element execution time is greater than μ, and t, $0 < t < 1$, corresponds to the fraction of T concentrated of the portion of the array identified by d. Therefore, it follows that $t > d$. 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 of the portion of the array identified by d. 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 $F = \frac{t}{d}$. F is directly proportional to the unbalanced workload that characterizes the array.

We implemented our template on a 64 node Cray-T3D by using MPI, release CRI/EPCC 1.2a. The resulting performances depend on some parameters. The factor of imbalance F and the per-iteration *average* execution cost μ both depend on the problem instance. The *Threshold* and the chunk size g, i.e. the constant number of iterations included in each chunk, must be instead tuned on the basis of machine features and of an approximate knowledge of μ and F.
Figure 2 shows the actual execution times obtained on problem instances characterized by $F = \{1, \ldots, 9\}$ and $\mu = 0.15$ *msec*, $\mu = 0.3$ *msec* or $\mu = 0.6$ *msec*. The plots compare the template execution times with

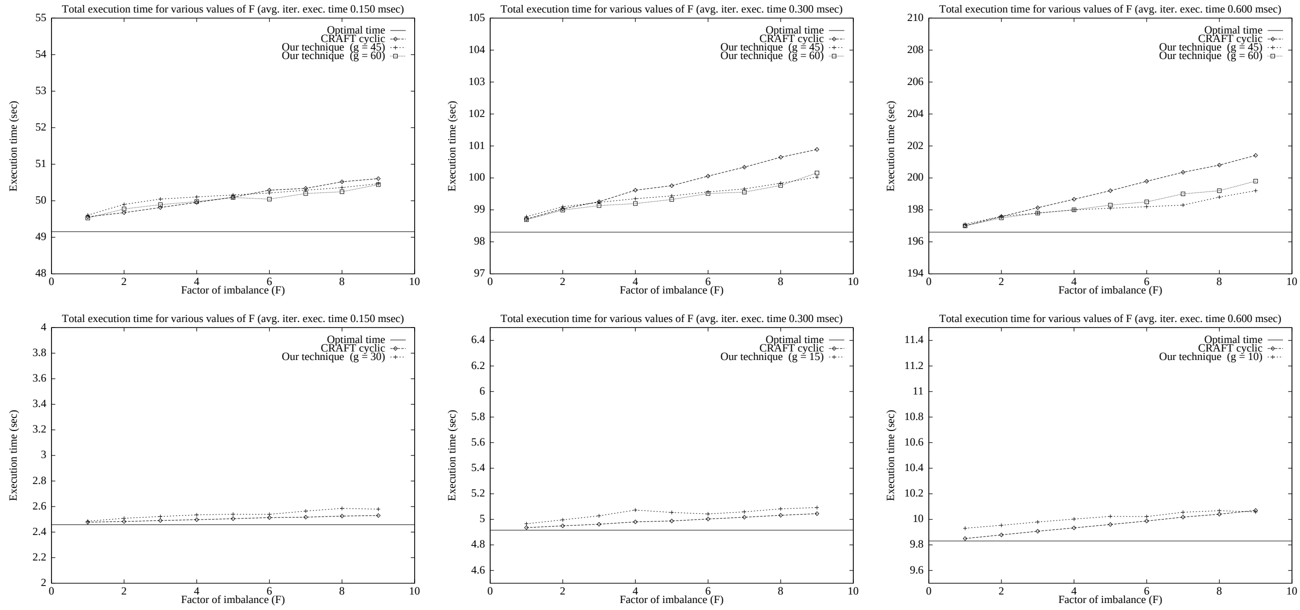

Figure 2: Execution times for different values of F, μ, K, and comparison with the optimal execution times and the execution times of the CRAFT-Fortran program adopting a cyclic array distribution.

the theoretical optimal execution time, and with the execution time obtained by the CRAFT-Fortran version of the same problem instance exploiting a cyclic array distribution[1]. The first three plots are related to the template for *iterated parallel loop* with $K = 20$, while the other three plots refer to the template exploiting partial data replication. As it can be seen, the execution times obtained by our template are always better than CRAFT-Fortran *cyclic* ones when the workload is unbalanced ($F > 3$) and the loop is iterated, while are comparable with the CRAFT-Fortran version when no imbalance occurs or the parallel loop is executed only once. Moreover, our template does not require the programmer to choose the best data distribution, since we adopt a *blocking* data mapping which, depending on the program features, may be very useful as it allows locality to be effectively exploited. The possibility of exploiting data locality is one of the main goals of our technique, even if the benchmark we used to validate the template does not take advantage of it.

References

[1] R. Alverson et al. The Tera computer system. In *Proc. of the 1990 ACM ICS*, pages 1–6, 1990.

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

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

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

[5] 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.

[6] E.P. Markatos and T.J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. *IEEE TPDS*, 5(4):379–400, April 1994.

[7] 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 ICS*, 1996. to appear.

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

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

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

[1] The times obtained by the CRAFT-Fortran version when a *blocking* distribution is adopted are about F times the optimal ones. For example, for $K = 20$, $F = 9$ and $\mu = 0.3$ *msec*, the execution time of the *blocking* CRAFT-Fortran program is 895 seconds w.r.t. less of 100 seconds of our technique.