Using preemptive thread migration to load-balance data-parallel applications

Gabriel Antoniu, Christian Pérez

Using Preemptive Thread Migration to Load-Balance Data-Parallel Applications

Gabriel Antoniu and Christian Perez

LIP, ENS Lyon, 46, Allée d'Italie, 69364 Lyon Cedex 07, France
{Gabriel.Antoniu, Christian.Perez}@ens-lyon.fr

Abstract. *Generic* load balancing policies for irregular parallel applications may be efficiently implemented by integrating *preemptive* thread migration into the runtime support. In this context, a delicate issue is to manage pointer validity in a migration-safe way. In [1] we presented an *iso-address* approach to this problem. This paper discusses the impact of the iso-address migration scheme on the runtime of the Adaptor [4] HPF compiler. This runtime (previously modified so as to generate multithreaded code for our PM2 runtime system [3]) now provides a generic support for *dynamic* load balancing, using preemptive thread migration. We report some encouraging results obtained with our system on a HPF flame simulation code, a motivating application of HPF 2.0 [7].

1 Introduction

Load balancing is a key issue for high performance parallel distributed computing. When the application behavior is hardly predictable at compile time, *dynamic* load balancing becomes essential. This can be achieved either by redistributing the data, or by migrating the activities across the available nodes. The former method may be very efficient, but it is generally highly dependent on the application. Besides, data redistribution code has to be explicitly inserted into the application code, at well-defined points so that a good tradeoff between responsiveness to load imbalance and overhead is difficult to obtain. The latter method detects load imbalance and responds by migrating activities from the overloaded nodes to the underloaded ones, provided activity migration is preemptive. In this approach, generic balancing policies may be implemented *outside* the application code and transparently applied to the application activities. This approach may be less efficient than data redistribution if activities are implemented as regular, heavy processes. In this context, threads have proven adequate as a means to accomplish fine-grained load balancing thanks to their ability to be efficiently migrated from one processor to another.

The framework of our study is data-parallel compiling for distributed-memory systems. In the approach we present in [3], the code dedicated to load balancing is separated from the HPF [8] application code. HPF abstract processors are projected onto threads. The load is balanced by transparent thread migration. To achieve this, *preemptive* migration is necessary. However, the migrating threads

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 117–124, 1999.

may execute arbitrary codes, in particular they may handle pointers to stack data or to allocated heap data. This is a constraint with respect to preemptive migration, since these pointers may become invalid when the threads get moved to another node. In [1] we propose a solution to this problem, based on an *iso-address* allocation scheme which guarantees that all pointers are migration-safe. This allocator was integrated into our PM2 [12] multithreaded environment, which serves as a runtime support for the Adaptor HPF compiling system [3]. To benefit from the iso-address approach, we have modified the compiler runtime and then tested the whole compiling system using a flame simulation code [7].

The remainder of this paper is organized as follows: in Section 2 we give a brief description of our iso-address allocation scheme. Section 3 shows how the Adaptor HPF compiler runtime was modified to integrate the iso-address allocator. Section 4 describes our experiments with the flame simulation code and presents some performance figures. Section 5 offers a few concluding remarks.

Related Works

Thread migration is provided by several multithreaded systems, the underlying motivations being different from case to case. In Ariadne [10], thread migration is the only way to access remote data. In Millipede [9], it is used to balance the load. In both systems, data are shared. Our study is concerned with a different case, in which data are not shared by multiple threads: each piece of data belongs to some unique thread and has to follow it on migration.

Migrating threads dealing with pointers is a common problem for all systems providing thread migration [6, 9, 10, 15]. Nevertheless, the solutions proposed are not satisfactory with respect to both efficiency and flexibility. In Ariadne [10], thread stacks are relocated at a usually different address on the destination node, such that pointers need to be updated. As shown in [1], several important problems cannot be solved by this approach. In Millipede [9], threads and their data are always relocated at the same virtual addresses on all nodes. Yet, thread creation is expensive, therefore the number of concurrent threads is statically fixed at initialization. In UPVM [5], threads are provided with private heaps whose sizes are fixed at thread creation, so the amount of data that a thread can allocate is limited. Also, UPVM thread creation is expensive, since it is requires a global synchronization. The approach we propose in [1] avoids the static limitations while keeping efficient and provides an appropriate support for building migration-based strategies for dynamic load balancing (see Section 4).

2 Iso-Address Allocation: A Sufficient Condition for Preemptive Migration

Preemptive migration relies on the ability to interrupt a running thread *at an arbitrary point of its instruction stream* and to move its resources (i.e. its stack, descriptor and private data) to another node, where it can resume its execution. A difficulty arises as soon as the migrating thread uses pointers to access

data stored in its stack or in the heap of the host process. Two kind of pointers are concerned: *user-level* pointers, explicitly declared in the applications, and *compiler-level* pointers, implicitly generated by the compiler. For instance, frame pointers, used to link stack contexts, belong to this latter class, as well as auxiliary pointers, generated in order to avoid indirections when accessing data. Moreover, pointers may be present in registers when the thread gets interrupted. The migration mechanism must ensure the validity of all these classes of pointers and must guarantee their safe use after migration.

2.1 The Iso-Address Approach

As shown in [1], the best approach to the problem is to design a mechanism which ensures that a thread and its data can always migrate while keeping the same virtual addresses. Our iso-address memory allocator (called `isomalloc`) guarantees that each virtual address range allocated on a node is kept free on any other node, such that iso-address migrations generate no overwriting. After such an operation, the thread may resume its execution without any post-migration processing, since all pointers remain valid.

A simple, but expensive way to ensure this global consistency would be to reserve a virtual address range on all nodes each time an allocation is performed, by means of a global synchronization. A more efficient strategy is proposed by `isomalloc`, allowing each node to locally allocate memory within a set of reserved address ranges without informing the other nodes. A special area in the virtual address space is dedicated to iso-address allocations on all nodes (`iso-address area`). This area is divided into a series of fixed-size ranges called *slots*. Each slot is assigned to a single node and, conversely, each node may *locally allocate* memory into its *globally reserved* slots without informing the other nodes.

2.2 Managing the Iso-Address Area

At any moment, a slot belongs either to a node, or to a thread. Only the slot owner is entitled to use the slot for memory allocation/release (internally carried out through calls to the `mmap`/`munmap` primitives) and to access it in read or write mode. This means that the data stored in a given slot belongs to a unique owner, since *slots are not shared*. At application startup, each slot is assigned to some node according to some arbitrary, user-specified distribution. When a thread gets created, it "buys" a slot from the local node, to store its stack. If the thread dynamically allocates iso-address memory, it gets other slots from the local node. On migration, all these slots automatically migrate along with the thread. Finally, when the thread releases its memory, the corresponding slots are given to the current local node, which may be different from their initial owner.

A particular problem arises when a thread allocates a block larger than the largest iso-address region (i.e. set of contiguous slots) owned by the local node. In this case, the latter may "buy" slots from the other nodes by means of a

global negotiation. This operation is obviously expensive and should not occur too frequently. Therefore, a "smart" initial slot distribution (which may be application-dependent) and an appropriate slot size should be chosen, in order to obtain an efficient execution. In our implementation, the slot size is fixed to 16 memory pages, which is large enough to contain the initial resources of a thread (i.e. its stack and its descriptor). Consequently, thread creation is a local operation irrespective of the slot distribution, since a single slot is required for the thread resources.

2.3 Using `isomalloc`

The PM2 programming interface provides two primitives for iso-address allocation/release operations `pm2_isomalloc` and `pm2_isofree`, with the same prototype as the `malloc` and `free` primitives.

Threads should call `pm2_isomalloc` instead of `malloc` to allocate memory for data which must follow the thread on migration. Any area allocated through `pm2_isomalloc` should be released via `pm2_isofree`. These primitives guarantee that the references to the corresponding memory areas are kept valid in case the thread migrates to another node. Consequently, threads may use pointers and still be preemptively migrated. Notice that `pm2_isomalloc` and `malloc` are not incompatible: the `malloc` primitive may still be used to allocate memory for non-migratable data.

3 Impact on the Runtime of a Data-Parallel Compiler

3.1 Integrating `isomalloc` in a HPF Compiler Runtime

As explained in Section 1, we use PM2 as a runtime support for the Adaptor [4] HPF compiler. The compiler runtime was previously modified in order to generate multithreaded code for PM2 [3]: each HPF abstract processor is mapped onto a thread. We have updated the runtime of Adaptor 4.0a, in order to take advantage of the iso-address allocator.

In the runtime version without `isomalloc`, an abstract processor had to explicitly call the migration function when it decided to migrate. The main objective of this function is to pack all data having to follow the migrating abstract processor: *dynamically* allocated data and static data specific to the Adaptor runtime. When arriving at the new node, a symmetric unpack function is invoked. Then, in order to build the environment of the abstract processor as it was on the original node, many calls to the `malloc` function are needed.

In the runtime version with `pm2_isomalloc`, only the static data need to be packed and unpacked. All memory areas dynamically allocated by `pm2_isomalloc` are automatically migrated with the abstract processor. The iso-address allocator has thus allowed a simplification of the runtime, while the migration time for abstract processors remains basically the same.

Preemptive migration has introduced critical sections in the program. The critical sections are located in the communication part of the runtime because

that part of the code accesses shared data, such as the list of waiting messages for a thread.

A last point related to preemptive migration is the way the location of an abstract processor is managed. If an abstract processor $ap1$ located on process $p1$ needs to send a message to an abstract processor $ap2$, it has to know on which process $p2$ that abstract processor is located. Our implementation does not need accurate knowledge about abstract processor location. Every message finds its addressee thanks to a message forwarding mechanism. Optimization techniques, such as regular broadcast of abstract processor localization, can be used to avoid a long chain of forwarding.

4 Application Study: A Flame Simulation Code

4.1 Benchmark Description

To evaluate the benefits of preemptive abstract processor migration, we have chosen the flame simulation code (illustrated in Figure 1), which is one of the motivating applications of HPF 2 [7]. The code performs a detailed time-dependent, multi-dimensional simulation of hydrocarbon flames in two phases. The first phase requires communications between neighbor mesh points. The computation load is the same in each point of the mesh. The second phase does not involve communications but the computational cost of each point varies as the computation progresses.

```
!lb$ begin load balancing with work stealing algorithm
    do time = 1, timesteps
C       convection phase
        x(2:NX-1,2:NY-1) = x(2:NX-1,2:NY-1) + F(z(2:NX-1,2:NY-2),
  $          y(1,NX-2),y(3,NX-1),y(2:NX-1,1:NY-2),y(2:NX-1,3:NY))
        y=x
C       Reaction phase
        forall (i=1:NX, j=1:NY) z(i,j) = Adaptive_Solver(x(i,j))
    end do
!lb$ end load balancing
```

Fig. 1. Flame simulation kernel code.

This application is interesting because the two phases have different requirements. The first phase is well suited for a block distribution (it requires neighborhood communication and the computations are regular), whereas the second phase needs an irregular distribution of the data.

4.2 Load Balancing Issues

The block data distribution minimizes communications in the first phase. But, in the second phase, the application suffers the effects of the load imbalance.

The cyclic data distribution has opposite effects compared to the block distribution. The load is quite well balanced in the second phase, but the communications are expensive in the first phase, given that each node, at each iteration, has to send all its data to its 2 (resp. 4) neighboring nodes if one (resp. two) dimension(s) is (are) cyclicly distributed. A more important drawback is the huge waste of memory. To receive data from the neighboring nodes, a node has to allocate 2 (resp. 4) arrays as large as the main array. So, only 33 % (resp. 20 %) of the memory available on each node can be used for local data.

Our approach consists in managing the mapping of HPF abstract processors onto the nodes. Using a block distribution and a regular mapping of abstract processors, best performance can be obtained for the first phase. To handle load imbalance in the second phase, we only have to migrate abstract processor from overloaded nodes to underloaded nodes. Since the load distribution is unknown, abstract processors are migrated according to a work stealing algorithm [2].

4.3 Performance Results

The experiments have been made on a 4-node cluster of 200 MHz PentiumPro PC connected by a Myrinet network [11] and using PM2 on top of the MPI-BIP communication library. The latency in this configuration is 25 μs and the bandwith is 120 MB/s.

We have considered three data sets, which differ in their degree of irregularity for the reaction phase. They range from a regular to a highly irregular computational pattern. For each data set, we first present the execution times obtained with the `BLOCK` and `CYCLIC` distributions, using the original Adaptor compiler. Then, we present the times obtained by `BLOCK`-distributing the data among 64 abstract processors on the 4 nodes, without and with load balancing.

Mode	Regular	Medium irregularity	High irregularity
block	3.69	5.34	9.51
cyclic	5.37	4.34	5.34
block + threads	3.36	3.10	5.32
block + threads + work stealing	3.57	2.85	4.45

Table 1. Time in seconds for different distribution on the flame simulation benchmark on four processors. The grid is 1024x1024 elements.

The experimental results are displayed on Figure 1. We can see that the cyclic distribution incurs severe overhead due to communication in all cases. The block distribution performs well when the computational cost is regular but its performance drops when it becomes irregular. Introducing several abstract processors per node allows communication to be overlapped by computation. It leads to better performance for the block distribution. Load balancing the application by migrating abstract processor according to a work stealing algorithm leads to

good performance in all circumstances. When the load is balanced, this strategy does not generate a significant overhead. When the load is not balanced, abstract processor migration improves performances.

4.4 Discussion

Orlando and Perego [14] have studied this application in depth. They hand-coded on top of MPI different solutions. They found that the best performance is obtained when using dynamic load balancing techniques through the SUPPLE support [13]. SUPPLE is a run-time support for the implementation of uniform and nonuniform parallel loops. It adopts a static block distribution but, during the execution, chunks of iterations and associated data may be dynamically migrated towards underloaded processors. Migration is decided at run-time with a work stealing algorithm.

Our approach for this application is very similar. However, there are two main differences. First, our work is fully integrated into an existing HPF compiler, as detailed in [3]. Second, we have separated the execution of the HPF abstract processors from their location. The key point to achieve this is that we embed an HPF abstract processor into a migrantable thread. Thus, we do not have to care about the scheduling of computation and communication, as in the the SUPPLE support. Our approach is not limited to a specific kind of loop. So, we can claim that we have obtained an HPF compiler and runtime that can apply work stealing techniques to *any* HPF program. Nevertheless, the ability to dynamically move abstract processors makes us think that any load balancing strategy could be implemented in our framework.

5 Conclusion and Future Work

This paper studies how preemptive thread migration can be used to load-balance data-parallel applications. The major problem of preemptive thread migration lies in the management of pointers. Our iso-address allocator provides a simple and efficient solution to this problem. Its integration into a multithreaded version of the Adaptor runtime has been easy and has simplified the runtime. Last, the cost of abstract processor migration has not been significantly modified by this new functionnality. To validate the usefulness of preemptive abstract processor migration, we have benchmarked a flame simulation kernel code, which is part of the motivating applications of HPF-2. Previous research [14] working on this application with MPI-based hand-coded program has shown that the cyclic distribution may have good results. Since our solution based on a work stealing strategy leads to globally better performance than both the block and the cyclic distributions, we can conclude that preemptive thread migration is an efficient way to load-balance this kind of HPF applications.

The future developments of this work have two main directions. On one hand, we intend to study the relationship between HPF programs and load balancing strategies. Now that our modified version of Adaptor 6 is operational, we can

test HPF 2 distributions like the `CYCLIC(N)` or the `INDIRECT` distributions. On the another hand, it seems interesting to study the links between the iso-address allocator and distributed shared memory systems (DSM). The iso-address allocator seems to offer the basic functions to build such a system.

References

[1] G. Antoniu, L. Bougé, and R. Namyst. An efficient and transparent thread migration scheme in the PM2 runtime system. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99)*, Lect. Notes in Computer Science, San Juan, Puerto Rico, April 1999. Springer-Verlag. Workshop held as part of IPPS/SPDP 1999, IEEE/ACM.

[2] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, pages 356–368, Santa Fe, New Mexico, November 1994.

[3] L. Bougé, P. Hatcher, R. Namyst, and C. Perez. Multithreaded code generation for a HPF data-parallel compiler. In *Proc. 1998 Int. Conf. Parallel Architectures and Compilation Techniques (PACT'98)*, ENST, Paris, France, October 1998.

[4] Th. Brandes. *Adaptor*. Available at `http://www.gmd.de/SCAI/lab/adaptor/adaptor_home.html`.

[5] J. Casas, R. Konuru, S. W. Otto, R. Prouty, and J. Walpole. Adaptive load migration systems for PVM. In *Proc. Supercomputing '94*, pages 390–399, Washington, D. C., November 1994.

[6] D. Cronk, M. Haines, and P. Mehrotra. Thread migration in the presence of pointers. In *Proc. Mini-track on Multithreaded Systems, 30th Intl Conf. on System Sciences*, Hawaii, January 1997.

[7] HPF Forum. HPF-2 Scope of activities and motivating applications, November 1994. Ver. 0.8.

[8] HPF Forum. *High Performance Fortran Language Specification*. Rice University, Texas, Oct. 1996. Version 2.0.

[9] A. Itzkovitz, A. Schuster, and L. Shalev. Thread migration and its application in distributed shared memory systems. *J. Systems and Software*, 42(1):71–87, July 1998.

[10] E. Mascarenhas and V. Rego. Ariadne: Architecture of a portable threads system supporting mobile processes. *Software: Practice & Experience*, 26(3):327–356, March 1996.

[11] Myricom. Myrinet link and routing specification. Available at `http://www.myri.com/myricom/document.html`, 1995.

[12] R. Namyst and J.-F. Méhaut. PM2: Parallel multithreaded machine. a computing environment for distributed architectures. In *Parallel Computing (ParCo '95)*, pages 279–285. Elsevier Science Publishers, September 1995.

[13] S. Orlando and R. Perego. SUPPLE: an efficient run-time support for non-uniform parallel loops. Research Report CS-96-17, Dip. di Matematica Applicata ed Informatica, Universita Ca'Foscari di Venezia, December 1996.

[14] S. Orlando and R. Perego. A comparison of implemantation strategies for nonuniform data-parallel computations. *J. Parallel Distrib. Comp.*, 52(2):132–149, March 1998.

[15] B. Weissman, B. Gomes, J. W. Quittek, and M. Holtkamp. Efficient fine-grain thread migration with Active Threads. In *Proceedings of IPPS/SPDP 1998*, Orlando, Florida, March 1998.