EXPERIMENTS IN LOAD BALANCING ACROSS THE GRID VIA A CODE TRANSFORMATION

Eric Violard, Romaric David, Benjamin Schwarz
LSIIT-ICPS, CNRS-Université Louis Pasteur
Boulevard S. Brant, F-67400 Illkirch, France
{violard,david,schwarz}@icps.u-strasbg.fr

Abstract

We propose a code transformation to adapt a parallel MPI application to the grid. It aims at balancing the computational load across the processors in order to reduce the global execution time. This transformation may be applied to a rather wide range of parallel codes. It was originally designed for a Vlasov equation solver, which is particularly challenging due to the dependencies scheme it involves. Experimental results show the advantage of our code transformation compared with others system support approaches. This work is part of the TAG project.

Keywords: Grid computing, Parallel MPI application, Plasma Physics, Load balancing, Code transformation.

1. Introduction

Reducing global execution time of applications on a computational grid is a well-known problem that has often been addressed. Among the techniques handling this problem two are noticeable: *scheduling* and *load balancing*. Scheduling allows to determine which resources of the grid are to be used. The AppLeS project [10] mainly relies on the scheduling technique and demonstrates its interest. Load balancing tries to reduce the global execution time by assigning statically or dynamically more workload to the more powerful resources.

Our purpose in the TAG [1] project is to propose techniques and tools for the end user so as to improve the performances of parallel applications on computational grids. We thus work on both scheduling and load balancing techniques. Our project is original in the sense that we use source code transformations to see how it impacts on the performances. The chosen approach for designing relevant code transformations is to handle real applications written with MPI and validate code transformations on a real test grid.

In this paper, we propose a code transformation to enhance load balancing of parallel MPI applications. This transformation may be applied to a rather wide range of parallel codes. It was originally designed for a Vlasov equation solver, which is particularly challenging due to the dependencies scheme it involves. Experimental results show the advantage of our transformation compared with others system support approaches.

Our technique relies on a static load balancing approach as opposed to dynamic load balancing. Static load balancing attempts to determine appropriate shares of work to be distributed to processors at application startup, while dynamic load balancing may redistribute part of the work during the execution to compensate for differences in processors performance.

Classical dynamic load balancing strategies [9] seem well suitable for grid environment as computing and network resources always change. However, such dynamic load balancing strategies may fail because of the many redistributions they imply. Moreover, data dependencies may slow down fastest processor to the slowest one [3]. Static strategies may be very useful when the speeds of the processors are known.

The outline of the paper is as follows: we first present the application in Plasma Physics which was used to investigate the load balancing problem. It is presented here as an illustration for introducing our code transformation. Then, we describe this transformation and define the class of applications it can be applied to. Finally, we present experimental results obtained on a test grid in order to validate our transformation.

2. A motivating example

Our example is an application devoted to the numerical simulation of problems in Plasma Physics and particle beams propagation. Among these problems is the study of controlled fusion, which seems to be a promising solution for future energy production. The set of particles is described by a particle density $f(t, x, v)$, depending on the time t, the position x, and the velocity v. The evolution of the particles is modeled by the Vlasov equation whose unknown is function f. The application implements the PFC resolution method [5] discretizing the Vlasov equation on a mesh of *phase space* (i.e. position and velocity space).

The values of the distribution function f at a given time t^n are stored in a (large) matrix $(F^n_{i,j})$ where i represents the index of position and j, the index of velocity on the space phase mesh.

The PFC method defines the values of F^{n+1} from the values of F^n by introducing an intermediate matrix $F^{(1)}$ of same size. The PFC method and its parallelization has been presented in [6].

The parallel code is obtained by using the data decomposition classical technique. Assuming P identical processors are available, matrices are split into $P \times P$ blocks of equal size. Each processor owns P blocks on a row of F^n and is responsible for the computation of P blocks on a row of F^{n+1}. The code for each processor then consists at each time step in computing local blocks of $F^{(1)}$ from local blocks of F^n (this does not require any communication), communicating blocks of $F^{(1)}$ in order to perform a global transposition of matrix $F^{(1)}$, computing blocks of ${F^{n+1}}^T$ from the received blocks of ${F^{(1)}}^T$ (without any communication), and communicating blocks of ${F^{n+1}}^T$ in order to perform a global transposition of matrix ${F^{n+1}}^T$. These operations are ordered to overlap communication with computation. As illustrated in [6], overlapping is required as transposition is very time-consuming.

Maximizing overlapping implies a good scheduling of the computation of the blocks on each processor. A block is asynchronously sent as soon as it has been computed and a block on the diagonal is computed last, as shown in figure 1. Before the next time step, processes synchronize in order to wait for all the blocks to be received.

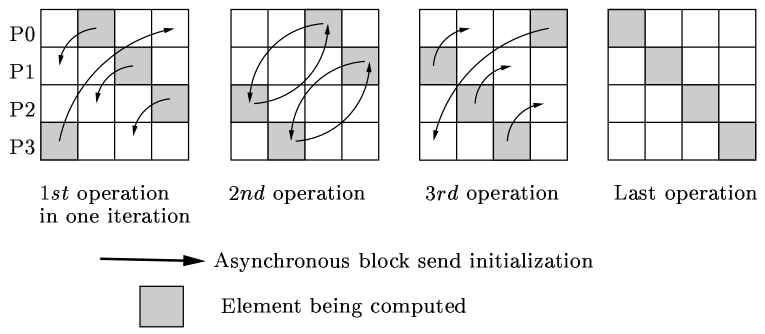
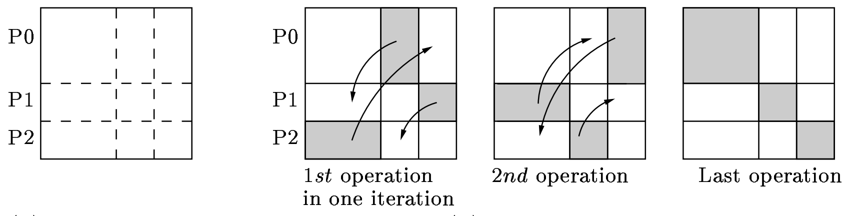


Figure 1. Matrix transposition in 4 operations with 4 processors

This code has been written for an homogenous parallel machine. We now consider a simple adaptation of it to an architecture with heterogeneous processors, such as a grid.

Since computation time depends on the amount of data to be computed on a given processor, one classical solution to achieve load balancing is to redistribute data according to the relative power of the available processors. This would have been possible in the present code, by parametrizing the data allocated to each processor, i.e., the amount of data allocated to a processor depends not only on the size of the matrix and the number of processors, but also on the relative power of the processor. For example, on figure 2(a), we illustrate data repartition of a 4x4 matrix on three processors, one of them being twice as powerful as the others.

This code adaptation, illustrated on figure 2(b) without any further transformation would have led to an unbalanced execution time on each processor. Indeed, as the computation is achieved in the previously described order (figure 1), overlapping is lower because blocks are sent at different times by the processors. In the homogeneous process described above, blocks are sent concurrently on each processor, thus maximizing overlapping.



(a) Matrix splitting (b) Unbalanced transposition

Figure 2. Matrix splitting and transposition with an unbalanced data distribution

This example is representative of a class of codes whose data distribution pattern is dictated by the algorithm and for which a naive data redistribution is not sufficient to reach an acceptable load balancing. It is therefore necessary to investigate some other solutions which can preserve the data distribution pattern. In such applications where computation can still be divided into any number of processes, a naive solution to achieve load balancing at runtime is to assign more processes to the more powerful processors. For instance, given 2 processors, one being 1.5 faster than the other it is possible to perform a load balance by executing 3 processes on the fastest machine, 2 on the other one.

Such a procedure implies a lot of system overhead, due to inter-process communications and time-sharing mechanisms. An idea to avoid this, is to rewrite the code in such a way that only one process would be launched on each processor. This process will compute all data given to the n processes we intented to run on the given processor, i.e., will emulate or serialize the parallel execution of the n processes. We followed this idea to define a code transformation applicable to such applications.

3. The code transformation

We define our code transformation in two steps: first, we give the characteristics that the parallel code must have, then we list the changes that have to be done into the code.

3.1 Assumptions

Our transformation applies if the following assumptions are verified:

Assumption 1 *The code is SPMD.*

Assumption 2 *The code works with any number of processes, and for any given number of processes the workload is evenly spread over the processes.*

The workload is the amount of work, i.e., a mixture of the algorithm complexity and the data amount. It results in a given computation time on each processor. If the algorithm complexity is a linear function of the data size, then workload spreading is equivalent to data spreading.

Assumption 3 *The number p of available processors at a given time is known as well as their speed ratings. We note $v_i \in \mathbb{Q}^{+*}$, the normalized measurement of the speed of processor number i.* For instance, on a grid composed of 2 processors, one being 2.5 times quicker than the other, we write $p = 2$, $v_1 = 1$ and $v_2 = 2.5$.

The measurement of processors speed can be obtained with Spec (Standard Performance Evaluation Corporation) results or by benchmarking the processors with the application. We use a normalized SpecRate, where the less powerful processor is rated 1.

3.2 Changes into the code

The transformation is sketched on figure 3. As said previously, it consists in emulating several processes on a single one. From assumption 1, the initial code of all processes falls into successive computation and communication sections with calls to MPI communication functions (arrow (a) of figure 3). The code for the single process can then be built by combining these sections into a single code (arrow (b) of figure 3). The computation sections are put together and similarly for the communication sections. The MPI calls in the communication sections have to be changed in order to map emulated process onto real processes and perform memory copy instead of communications when sender and receiver are on the same node.

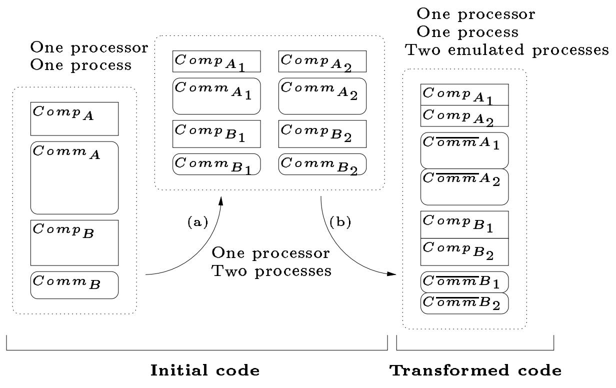


Figure 3. Code transformation

Moreover, blocking point-to-point communications such as `MPI_Send` have to be replaced by their non-blocking versions (e.g. `MPI_Isend`) in order to avoid deadlock situations by letting the system handle communications. Calls to collective communications have to be performed only once on a processor, no matter the number of emulated processes on this node. Some collective communications such as `MPI_Barrier` do not require any other changes whereas some others do. For instance, a `MPI_Scatter` that spreads equally all datas from one node called root to all others should be replaced by a `MPI_Scatterv` that allows an heterogeneous spreading thus allocating chunks of data proportionnally to the number of emulated processes on each node.

Workload sharing. A *workload sharing* is defined by a vector $(n_i)_{i \in 1..p} \in \mathbb{N}^p$ where n_i is the number of emulated processes that should be allocated to the processor P_i, $i \in 1..p$ of the grid. We must provide the transformed code with a sharing that divides the workload so that the execution time will be approximately the same on all processors. This can be achieved by allocating to each processor P_i a number n_i of emulated processes proportional to v_i.

4. Experimental results

For our experiments, we use our code in Plasma Physics and a test grid made of 4 processors on two LAN linked by Fast-Ethernet and

ATM Links. On the first LAN, two PC's equipped with Athlon XP 1800+ processors are used. The second LAN contains an heterogeneous Origin 2000 (Mips R10k/195Mhz and R12k/300Mhz). We use Globus [8] and MPICH-G2 [7] to launch the MPI application on these machines.

We have obtained the speeed ratings by combining the SpecFP for these three kind of processors. It gives $v_{R10k}=1$, $v_{R12k}=2$, $v_{XP}=4$.

We have measured the wall-clock time of :

1 the initial application without load balancing, i.e., with the same amount of data on each node.

2 the initial application with a basic load balancing using the system to launch several processes to a single node. We launched one process on the R10k, two processes on the R12k, and 4 processes on each of the two XP.

3 the transformed application with 11 emulated processes according to the workload sharing defined by the speed ratings.

Note that, in the second case, we checked that, several processes truly run on one processor. On monoprocessor machines, like the PC's, this is obviously true. On the parallel machine Origin 2000, we had to modify Globus in order to make this feasible.

Experiments was done for 32, 48 and 64 points of discretization. Results are reported on table 1. We notice that time loss due to system overhead decreases as data size increases. The modified algorithm is always better than the non-modified one.

Size	No load balancing	System load balancing	Software load balancing
32^4	342	525	301
48^4	2005	2223	1874
64^4	5380	4752	4404

Table 1. Elapsed time (s)

5. Conclusion and future work

In this paper, we presented a transformation that enables some characteristics of the grid to be integrated into the parallel code, thus enabling load balancing at runtime. This static workload balancing shows promising results, as it is significantly more efficient than pure-system workload balancing, even if the performance model relying only on processor speed is quite simple.

An interesting property of our transformation is that it preserves the code structure. In particular, the communication scheme remains unchanged. This will allow us to dynamically redistribute data, according to processor availability or network bandwidth evolution during execution.

Interesting mesurements may consists in evaluating the loss of performance due to the transformation. Indeed, using a great number of emulated processes causes an algorithmic overhead. For example, we could point out the number of emulated processes that can be handled on a node without loss of performance. This overhead could then be reduced by modifying the communication scheme, grouping multiple sends into one for example. Also, an automation of the rewriting procedure could save developper's time.

Last, we advocate the use of specific directives as in HPF [4]. Directives can summarize code transformations and enable the user to modify its source code incrementally.

References

[1] TAG Project, *Transformation and Adaptation for the Grid,* `http://grid.u-strasbg.fr`

[2] I. Foster, C. Kesselman, The Grid, Blueprint for a New Computing Infrastructure, *Morgan Kaufmann Publishers, Inc.*, 1998, ISBN 1-55860-475-8.

[3] J. F. Mehaut, Y. Robert, Algorithms and Tools for (Distributed) Heterogeneous Computing: A Prospective Report, ENS-Lyon, LIP Research Report n 1999-36, August 1999.

[4] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., M. E. Zoselem, The High Performance Fortran Handbook, *Scientific and Engineering Computation*, January 1994.

[5] F. Filbet, E. Sonnendrücker, P. Bertrand, Conservative Numerical schemes for the Vlasov equation, *J. Comput. Phys.* 172 (2001), pp 166–187.

[6] E. Violard, F. Filbet, Parallelization of a Vlasov solver by communication overlapping, to appear in proceeding of PDPTA'2002, Las Vegas.

[7] I. Foster, N. Karonis, A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems, *Supercomputing*, November 1998.

[8] I. Foster, C. Kesselman, Globus: A Metacomputing Infrastructure Toolkit, *International Journal of Supercomputer Applications*, 1997, pp 115–128.

[9] V. Kumar, A. Y. Grama, V. N. Rao, Scalable Load Balancing Techniques for Parallel Computers, *Journal of Parallel and Distributed Computing*, 1994

[10] F. Berman, R. Wolski, S. Figueira, J. Schopf, G. Shao, Application-Level Scheduling on Distributed Heterogeneous Networks, *Proceedings of SuperComputing*, 1996