TAG research report

LSIIT-ICPS

Pôle API, Bd. S. Brant
F-67400 Illkirch

August 2, 2002

Source Code Transformations Strategies to Load-balance Grid Applications

Romaric David, Stéphane Genaud, Arnaud Giersch,
Benjamin Schwarz, and Eric Violard
{david,genaud,giersch,schwarz,violard}@icps.u-strasbg.fr
Tel: +33 3 90 24 45 42 / Fax: +33 3 90 24 45 47

August 2, 2002

Abstract

We present load-balancing strategies to improve performances of parallel MPI applications running in a Grid environment. We analyze the data distribution constraints found in two scientific codes and propose adapted code transformations to load-balance computations. We present the general framework for our load-balancing techniques. We then describe these techniques as well as their application to our examples. Experimental results confirm that adapted source code transformations can improve Grid application performances.

1 Introduction

With the growing popularity of middle-ware dedicated at making so-called *Grids* of processing and storage resources, network based computing will soon offer to users a dramatic increase in the available aggregate processing power. The end-user will benefit from such a computing power only if new tools are made available to ease the use of such computing resources. Traditionally, many scientific applications have been developed by users with a parallel computer in mind. Users are thus used to work on a single host with a single-image file system and most of the time, a unique memory space (even if it is physically distributed). Furthermore, the parallel code is designed assuming an homogeneous set of processors with homogeneous and very high bandwidth links between processors. All these features that make the user comfortable with the system disappear with clusters or Grids. There are numerous difficulties that should be overcome to make the use of Grids more popular.
One of the major difficulties lies in the poor performances obtained with most parallel applications when run on a Grid. There are two major reasons for the lack of performance: first, the processors available are heterogeneous and hence the work assigned to processors is often unbalanced, and secondly the network links are orders of magnitude slower on the Grid than in a parallel computer. Much work has been carried out to take into account such heterogeneous environments for distributed computing: these efforts range from the optimization of communication libraries to the migration of processes to balance processors load. Among these techniques, load-balancing is certainly one of the most useful. Studies on load-balancing generally consider the running processes and the physical resources they use, to decide how to load-balance (e.g. [BGW93]). Less often, load-balancing code is included into the application source code itself so as to improve performance in some particular cases (e.g. [SWB98]). However, few research works concern load-balancing strategies specifically guided by the application to be run. The well-known AppLeS project [BWF$^+$96] uses information drawn from a specific application to schedule the execution of the application processes, but they do not modify the application source code to improve its execution. Thus, our project is original in the sense that we study how **source code transformations** may impact on the performances obtained when running applications on Grids, and

eventually systematically operate these transformations so as to produce programs permanently adapted to heterogeneous environments. To validate our ideas, we work on some real scientific applications written with MPI.
This paper presents preliminary results concerning load-balancing techniques we designed and implemented to improve performances of two scientific applications. We analyze the constraints inherent to the applications and propose adapted code transformations to load-balance computations. The paper is organized as follows: the first section explains the general framework for our load-balancing strategies and the two following sections describe the two test applications. For each application, we first explain its requirements, which load-balancing may be used and under which hypotheses. We finish the sections with experimental results. The last section comments on the code transformations operated and discusses future work.

2 Study framework

Our work is part of the TAG project[1] which aims at proposing tools for the end user so as to improve the usability of Grids. One important step to reach this goal is to identify the key characteristics of Grid applications in order to design techniques that would enable to write parallel applications optimized for a Grid environment. In this study, we have rewritten some parts of the tested applications to introduce load-balancing.
Load-balancing is a natural way to address the problems arising when working in an heterogeneous environment. Many research works have been conducted in this domain, the greater part of which usually discern two strategies: *static load-balancing* and *dynamic load-balancing* [KK92]. Static load-balancing attempts to determine appropriate shares of work to distribute 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. In our work, we try to consider both alternatives and assess which would be more profitable depending on the application type.
Our study is limited to SPMD applications using a message passing programming model. To validate our ideas, we work on real programs such as the two scientific applications presented as motivating examples in the two next sections. The two applications differ by their data distribution constraints, and we have to encompass how this impacts on the load-balancing strategies to apply. The first application belongs to the so-called *embarrassingly parallel* class of applications, which means that data distribution is unconstrained: it is possible to send any chunk of data to any process. On the contrary, the second application exhibits data dependencies and hence the data distribution is constrained. These two different data distribution constraints motivated two distinct load-balancing answers detailed in the next two sections:

- for unconstrained data distribution, we propose modifications of the source code to allow a direct data repartition accordingly to the processors and network links speeds,
- for constrained data distribution, we propose a load-balancing technique consisting in emulating real MPI processes: on each machine the transformed code simulates the execution of more or less processes depending on the processors speed.

3 Load-balancing for unconstrained data distribution

3.1 Motivating example: a geophysical application

We consider for our first experiment a geophysical code in the seismic tomography field. For our purpose, we only need to roughly sketch one part of the application structure. The input data is a set of seismic waves, each described by a pair of 3D coordinates (the coordinates of the seism source and those of the receiving captor) plus the wave type. With these characteristics, a seismic wave can be modelized by a set of *ray paths*, that represents the wavefront propagation. Seismic

[1]TAG stands for Transformations and Adaptations for the Grid (`http://grid.u-strasbg.fr/`).

wave characteristics are sufficient to perform the ray-tracing of the whole associated ray path. Hence, all ray paths can be traced independently. It is also important to note that the ray-tracing of the various ray paths require a varying amount of computations. The parallelization of the application, presented in [GGM02], assumes that all available processors are homogeneous (the implicit target for execution is a parallel computer). The following pseudo-code outlines the main communication and computation phases:

```
if (rank = ROOT)
    raydata ← read $n$ lines from data file;
MPI_Scatter(raydata, $n/P$, ..., rbuff, ..., ROOT,MPI_COMM_WORLD);
compute_work(rbuff);
```

where P is the number of processes involved. The `MPI_Scatter` instruction is executed by the root process and the computation processes. In MPI, this primitive means that the process identified as `ROOT` performs a send of distinct blocks of n/P elements from the `raydata` buffer to all processes of the group while all processes make a receive operation of their respective data in the `rbuff` buffer. Figure 1 shows a potential execution of this communication operation, with p_1 as root process.

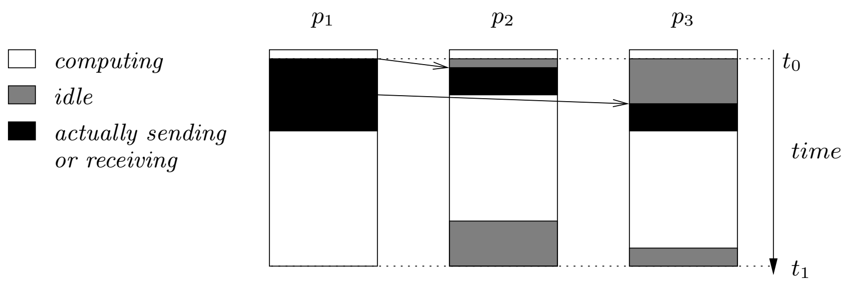


Figure 1: A scatter communication followed by a computation phase.

From this existing implementation, we now examine which transformations would be profitable for the execution on a Grid. We can quickly think of two manners of load-balancing this kind of application:

- The first one is *dynamic*: a classical master/slave scheme is a good candidate solution for such an heterogeneous environment. In this case, one master process reads small chunks of the input data set and distributes these chunks to slave processes, which compute the received piece of data. As soon as a slave process has finished its computations, it asks the master for a new chunk. The process repeats until the master sends a termination message to slaves.
- The second one is *static*: much like the above pseudo-code, the root process distributes the whole data set in a single communication round, except that it sends unequal shares of data whose sizes are statically computed on the basis of the processors and network performances.

We have implemented and tested both program transformations, and are particularly interested in comparing them in terms of performance but also in terms of code rewriting. As far as automatic program transformation is concerned, the second technique is more interesting, since MPI provides a `MPI_Scatterv` primitive that allows such irregular data distribution and hence the program transformation is more direct. Thus, we focus in our study on the conditions the application must meet to implement this load-balancing technique, and which performance results can be obtained with it.

3.2 Hypotheses for unconstrained data distribution

The static load-balancing technique applies for SPMD programs (also refered to as *data-parallel MPI* programs [Geo99]) that contain communication-computation phases. A communication-computation phase is characterized by a round of simultaneous communications between all processes, followed by local computations ended by a global synchronization —that is an explicit barrier or a synchronization through a second communication round. Note that programs which follow the *Bulk Synchronous Programming* model [Val90] are entirely build with successive such phases.
In the particular case of the geophysical application presented above, there is only one major communication-computation phase and only one process sends the whole input data set to other processes. Moreover, we must state further assumptions the programmer must verify before the load-balancing may take place.

(H1) *The data items in the input data set are independent.*

(H2) *The number of data-items is the only factor in time complexity.* Hence, giving any equal-size part of the domain to a given process will result in the same computation time.

3.3 Static load-balancing of computations and communications

We now introduce some basic metrics to compute the load-balancing: we need a metric to measure the time spent in a computation phase on a given processor during a given period, and a metric to evaluate the time spent to send a given volume of data during a given period. Given the changing state of network links and processors loads, we can ignore the periods at which events occur by using average values for computation and communication speeds. Our metrics thus only depend on data size. For a given communication-computation phase, we define:

- $comp_i(d)$ to be a continuous increasing function that returns the mean time needed to compute d data items on a processor p_i and,
- $comm_i(d)$ to be a continuous increasing function that returns the mean time spent in communication for d data items on a processor p_i.

Our objective is to minimize the elapsed time between the synchronization points ($t_1 - t_0$ on figure 1). Assuming (H2)[2], a process p_i among P processes should receive a share of d_i ($d_i \in \mathbb{N}$, $i \in [1, P]$) data items from the dataset of size D, such that:

$$max\{comp_i(d_i) + comm_i(d_i) \mid i \in [1, P] \text{ and} \sum_{i=1}^{P} d_i = D\} \tag{1}$$

is minimum. With these notations, the following problem formulation:

$$\begin{cases} comm_1(d'_1) + comp_1(d'_1) = \cdots = comm_P(d'_P) + comp_P(d'_P) \\ d'_1 + d'_2 + \cdots + d'_P = D \end{cases} \tag{2}$$

for which we search for the $d'_i \in \mathbb{R}$, implies that the $d_i = round(d'_i)$ satisfy equation (1), where the function *round* appropriately rounds the real values d'_i. This round process is described in Appendix B.2.

3.4 Load-balancing scatter operations

When the communication round is a scatter operation (as in our test application), we can precise the communication cost function and define it as the affine function: $comm_i(d) = b_i \cdot d + l_i$, where b_i and l_i are the network bandwidth and latency (resp.) from root processor to processor p_i.

[2]Note that without (H2) the problem would be far more complex: we would have to build all partitions of the data set in P subsets, and for each subset, evaluate the time of all the permutations from subsets to processors.

Moreover, (H2) implies that the computation cost function *comp*$_i$ is linear with the number of data items so we can let *comp*$_i$ be a linear function: *comp*$_i(d) = a_i \cdot d$.
Therefore, equation (2) can be written as the following linear system:

$$\begin{cases} (a_1 + b_1) \cdot d_1 + l_1 = (a_2 + b_2) \cdot d_2 + l_2 = \cdots = (a_P + b_P) \cdot d_P + l_P \\ d_1 + d_2 + \cdots + d_P = D \end{cases}$$

For shortness, let us define $c_i = a_i + b_i$ and $T = c_1 \cdot d_1 + l_1$. We thus write

$$\begin{cases} c_1 \cdot d_1 + l_1 = c_2 \cdot d_2 + l_2 = \cdots = c_P \cdot d_P + l_P \\ d_1 + d_2 + \cdots + d_P = D \end{cases}$$

For each i we can then write $d_i = \frac{T-l_i}{c_i}$ [3], which leads to $D = \sum_{i=1}^{P} \frac{T-l_i}{c_i} = T \times \sum_{i=1}^{P} \frac{1}{c_i} - \sum_{i=1}^{P} \frac{l_i}{c_i}$

And then have a general expression for T: $T = \frac{D+\sum_{i=1}^{P} \frac{l_i}{c_i}}{\sum_{i=1}^{P} \frac{1}{c_i}}$

From this we can extract a general expression for d_i:

$$d_i = \frac{D + \sum_{j=1}^{P} \frac{l_j}{c_j} - \sum_{j=1}^{P} \frac{l_i}{c_j}}{c_i \cdot (\sum_{j=1}^{P} \frac{1}{c_j})} \qquad (3)$$

Hardware model We must refine this expression with further assumptions on the networking capabilities of the nodes. We consider the full overlap, single-port model of [BCF$^+$01] in which a processor node can simultaneously receive data, send data to at most one destination (full-duplex network cards), and perform some computation. This model has shown to be realistic in our experiments, and is representative of a large class of modern machines.
As the root process sends data to processes in turn[4] a receiving process actually begins its communication after all previous processes have been served. The root process starts its computation after all the processes have received their data. This leads to a "stair effect" represented on figure 1 by the end times of the receive operations (black boxes). We call the delay during which a process p_i waits for the communication to actually begin, the *startup time*, noted s_i. If we let the root process be p_1, s_i is defined as:

$$\begin{cases} s_2 & = k \\ s_i & = s_{i-1} + b_{i-1} \cdot d_{i-1} \quad (i \in [3, P]) \\ s_1 & = s_P + b_P \cdot d_P \end{cases}$$

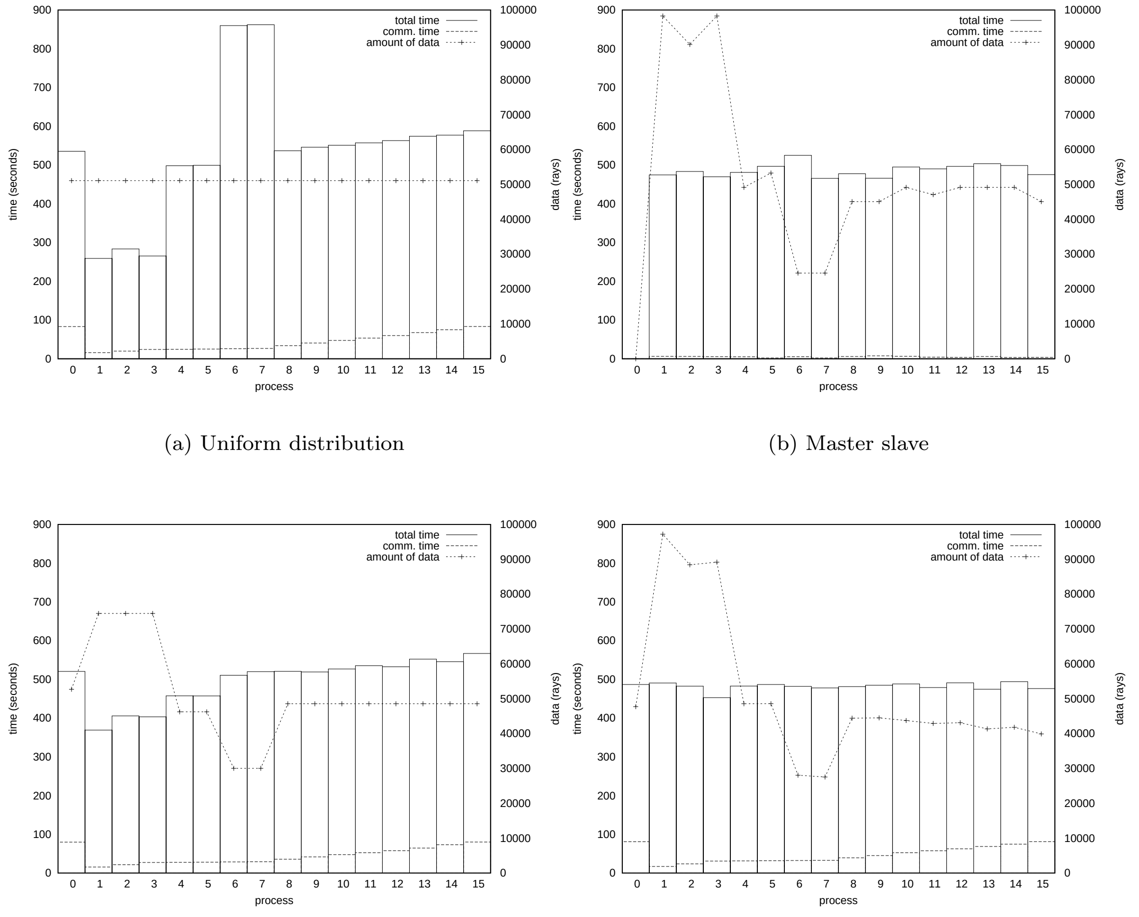
where k is a system overhead incurred by the preparation of data to be sent by the root process. The startup time should be integrated as part of the latency parameter (l_i) in equation (3).
The resolution becomes however not trivial because d_i depends on $d_1, \ldots, d_P$. A means to circumvent the problem is to omit, in a first instance, the latency to compute some initial $d_i = D/(c_i \cdot \sum_{j=1}^{P} \frac{1}{c_j})$. From these results we can approximate the startup times. These startup times can then be cumulated to latency to compute the final d_i.

3.5 Experimental results

Our experiment consists in the computation of 827,000 ray paths on 16 processors. Processors are located at two geographically distant sites. All machines run Globus [FK97] and we use MPICH-G2 [FK98] as message passing library. Table 1 shows the resource repartition across sites as well as the processors relative ratings. These ratings come from a series of benchmarks we performed on

[3] This expression as well as the following one are correct because the c_i are strictly positives.

[4] In the MPICH implementation, the order of the destination processes in scatter operations follows the processes ranks.



(a) Uniform distribution

(b) Master slave

(c) Load-balancing (without network)

(d) Load-balancing (with network)

Figure 2: Experimental results.

our application[5]. The table also shows the network bandwidths and latencies from node processor 0 that run the root process to any other processor. Hence, the communication speed from the root process to itself is considered infinite since memory copy delay is negligible as compared to network communication. These values have been measured either by running a ping application or by interrogation of a NWS daemon [WSH99] when available.
The first experiment evaluates performances of the original program in which each process receives an equal amount of data. Non-surprisingly, on figure 2(a), the processes end times largely differ, thus exhibiting an important imbalance. The second experiment shows the master/slave version behavior on figure 2(b), which appears well-balanced after we have finely tuned the size of data chunk sent to slaves. Note that there are only 15 computation processes in this implementation since the master process only handles data distribution.
Next, we experiment the load-balance of the scatter operation. As mentioned in the application description, the amount of computations needed for ray paths varies, which contradicts hypothesis (H2). However, given the numerous ray paths, we have been able to rearrange the data set so that rays are grouped in small blocks that require approximately the same amount of computations.

[5]On IRIX and Sun OS we used vendor supplied compilers and gcc on Linux. We arbitrarily chose a rating of 1 for the Intel 800MHz.

processes	0	1-3	4-5	6-7	8-15
sites	Illkirch1			Illkirch2	Montpellier
processor	PIII/933	Athlon/1800	PIII/800	R12k/300	R14k/500
processor ratings	1.14	1.61	1.00	0.65	1.05
bandwidth (MB/s)	$+\infty$	7.5		4	0.5
latency (seconds)	0	0.01		0.02	0.67

Table 1: A part of our test Grid with average performance measurements.

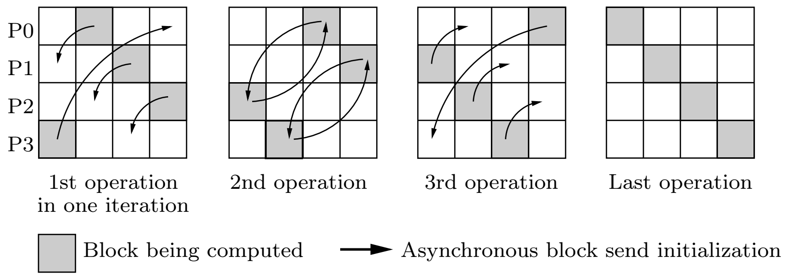
We can therefore use formulas from the previous section and measurements from table 1. We had to convert these measures in rays/second, and apply a corrective coefficient to fit as closely as possible to the application performances.
To assess important parameters, we have first tried to load-balance using the relative processors speed ratings only: figure 2(c) shows that omitting the network parameters leaves important imbalances and gives unsatisfactory results as compared to the master/slave implementation. The last experiment computes the load-balance using all parameters: figure 2(d) shows the best balance and confirms the importance of all parameters, especially to take into account the “stair effect” that clearly appears on figures 2(c) and 2(d). The total communication time in the master/slave execution is short as compared to the computation time. Consequently, most of the communication time measured in the scatter implementations actually is idle time. Only a small part of this time is spent in true communications of data. It is worth to note that the cumulated idle time (the surface of the “stair effect”) on figure 2(d) is approximately the computation time of one processor. This explains why with 15 computation processes, the master/slave implementation performs as well as the best load-balanced scatter with 16 computation processes. Therefore, the load-balanced scatter operation is efficient up to a a certain number of processors, but the master/slave model should outperform any scatter operation with many processors.

4 Load-balancing with constrained data distribution

4.1 Motivating example: an application in Plasma Physics

Our example is an application devoted to the numerical simulation of problems in Plasma Physics and particle beams propagation (e.g. 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 function $f(t, x, v)$, depending on the time t, the position x, and the velocity v. Its evolution is modeled by the Vlasov equation whose unknown is function f. The application implements the PFC resolution method [FSB01] 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 where $F^n_{i,j}$ represents the particle density of position at index i and velocity at index j. 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. Its parallelization has been presented in [VF02].
The parallel code is obtained by using a classical data decomposition 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 from 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 maximize the overlap of communications by computations. Figure

3 shows the scheduling of the computation of the blocks on each processor for $P = 4$. A block is asynchronously sent as soon as it has been computed and a block on the diagonal is computed last. Before the next time step, processes synchronize in order to wait for all the blocks to be received.



P0
P1
P2
P3
1st operation in one iteration
2nd operation
3rd operation
Last operation
Block being computed
Asynchronous block send initialization

Figure 3: Matrix transposition in 4 operations with 4 processors.

4.2 Hypotheses for constrained data distribution

This code has been written for an homogeneous parallel machine. We now consider the problem of tuning it to an architecture with heterogeneous processors. We therefore investigate a code transformation which can be applied in such a case. This transformation addresses the codes which have the following characteristics:

(H1) *The code works with any number of processors, and for any given number of processors, the workload is evenly spread over the processors (the code allocates an equal share of workload to each processor of a parallel machine).*

(H2) *The workload of any processor only depends on the amount of data to be processed (it does not depend on data values).*

(H3) *Data are structured and the computation of any datum depends on some other data in the same structure.*

(H4) *The way the data structure is distributed onto processors determines the cost of communications and there exists a best data distribution pattern to be preserved.*

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.

4.3 Load-balancing possibilities

We address programs for which the data distribution pattern is dictated by the algorithm. In this application the constraint lies in the requirement to transpose the matrix elements. Though we can not make a matrix element-wise redistribution as in the application of section 3.1 (hence the *constrained* data distribution), one possibility is to distribute matrix blocks whose size would depend on processors speed. However, as the blocks transfer durations are different for each processor in this case, overlapping of communication and computation is lower since send and receive operations are no more synchronized. In the homogeneous version described above, blocks are sent simultaneously on each processor, thus maximizing overlapping. It therefore seems worthwhile investigating some other solutions which can preserve the data distribution pattern.
In such applications where computations can be spread over any number of processes, a naive solution to achieve load balancing is to assign more than one process per processor. For instance, given 2 processors, one being 1.5 faster than the other it is possible to perform a load balance by

launching 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 real process would be launched on each processor. This process will compute all data given to the n processes we intended to run on the given processor, i.e. will *emulate* or serialize the parallel execution of the n processes. This idea is the basis of our code transformation.

4.4 Code modification

Our transformation is sketched on figure 4. As said previously, the transformation consists in emulating several processes by a single one. Since the initial code is SPMD, it decomposes into successive computation and communication phases. Communication phases mainly are sequences of calls to MPI communication functions. As the same code is executed by each process, any process of the initial program carries out one instance of the successive phases. Figure 4 on the left shows four phases of the initial program for two processes. Two such processes are emulated by a single process of the transformed code. Right hand side of figure 4 shows the content of the single process. Its code is built by combining the phases. The computation phases are put together and similarly for the communication phases. The MPI calls in the communication phases have to be changed in order to map emulated processes onto real processes and perform memory copy instead of communications when sender and receiver are on the same processor.

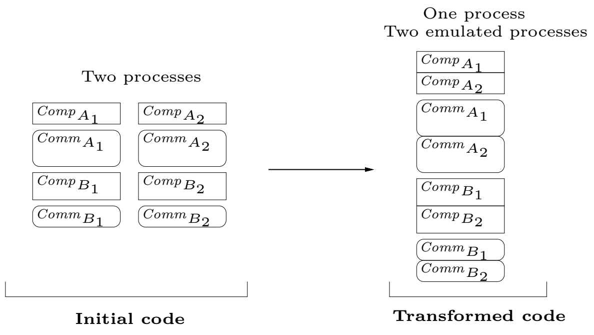


Figure 4: Code transformation.

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. One call to a collective communication has to be performed only once on a processor, no matter the number of emulated processes it holds. Some collective communications such as `MPI_Barrier` do not require any other changes. Some others do. For instance, a `MPI_Scatter` that spreads equally all data from one processor to all others should be replaced by a `MPI_Scatterv` that allows an irregular spreading so as to distribute chunks of data proportionally to the number of emulated processes on each processor.

Here is an example which focuses on the application of our program transformation on a piece of the code in Plasma Physics. For sake of simplicity, we only give *pseudo-codes*.

Initial pseudo-code of processor p_i ($i = 1..2$):

```
/* computation phase */
compute($A_i$);

/* communication phase */
MPI_Isend(..., $A_i$, $dest_i$, ...);
MPI_Irecv(..., $B_i$, $src_i$, ...);
```

This piece of code corresponds to the computation of one block of matrix and its transmission in order to perform the block transposition.

Transformed pseudo-code of processor p:

```
/* computation phase */
for ($i = 1..2$)
    compute($A_i$);
/* communication phase */
for ($i = 1..2$) {
    if (proc_of($dest_i$) $\neq p$)  MPI_Isend(..., $A_i$, proc_of($dest_i$), ...);
        else skip;
    if (proc_of($src_i$) $\neq p$)  MPI_Irecv(..., $B_i$, proc_of($src_i$), ...);
        else $B_i$ = $A_{src_i}$;
}
```

The communication phase examines the destination (resp. source) of the messages that have to be sent (resp. received) by the current emulated process. If the source and the destination emulated processes are embedded in the same process, then a memory copy is performed ($B_i = A_{src_i}$). If they are on distinct processes (proc_of($dest_i$) $\neq p$), then a real communication is performed between processes of rank proc_of($dest_i$) and proc_of($dest_i$).

4.5 Workload sharing

The transformed code is parameterized by a workload sharing provided at start up. 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 processor p_i, $i \in [1, P]$. We must provide the transformed code with a sharing that divides the workload so that the execution time will be roughly 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 its speed rating.

4.6 Experimental results

For our experiments, we use our code in Plasma Physics and a subset of our test Grid made of four processors (for $P = 4$): one R10k (referred to as p_1), one R12k (p_2) and two Athlon (p_3 and p_4).
Finding how many emulated processes are to be assigned to processors is a difficult problem. In a first instance, we approximate the number of emulated process from the speed ratings[6] of table 1. We choose n_1=1, n_2=2, n_3=4 and n_4=4 meaning for instance that an Athlon is rated four times better than a R10k. The results reported below in table 2 show significant gains in performance. Note however that work is under study to use theoretical results from [BDRV99] so as to find the optimal number of emulated processes. New experiments are currently undertaken to evaluate the extra gain that can be obtained with this method.
In order to validate our code transformation, we measured the wall clock time of:

1. The initial application without load balancing, i.e., with the same amount of data on each processor.
2. The initial application with a basic load balancing using the system to run several processes on a single processor. We spawn one process on the R10k, two processes on the R12k, and four processes on each of the two Athlon.
3. The transformed application with 11 emulated processes according to the workload sharing defined by the speed ratings.

In the second case, we checked that several processes truly run on one processor[7]. Experiments were done for 32, 48, and 64 points of discretization in each dimension of the phase space, which

[6]The R10k processor has a rating of 0.46.
[7]On monoprocessor machines, like the PC's, this is obviously true. On the parallel machines, we had to modify the Globus job-manager to make this feasible.

means that the total number of points of matrices F^n and $F^{(1)}$ was 32^4, 48^4, and 64^4 respectively (that is 16 MB, 80 MB, and 256 MB of memory). Results are reported on table 2. We notice that time loss due to system overhead decreases as data size increases. Modified algorithm is always better than non-modified algorithm, may there be or not system load-balancing.

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 2: Elapsed time (seconds).

5 Related work

The experimental study carried out in [SWB98] compares dynamic versus static load-balancing strategies in an image rendering ray-tracing application, executed on a network of heterogeneous workstations. The nature of the application is close enough to the geophysics application for us to benefit from their conclusions. They conclude that no one scheduling strategy is best for all load conditions, and recommend to investigate further the possibilities of switching from static to dynamic load-balancing. Our experiments confirm that static load-balancing requires precise information about parameters to be efficient whereas the master/slave model naturally adapts to heterogeneous conditions. Therefore, the variance of conditions could be taken into account to call one implementation or the other during execution. The idea to encapsulate parameterized code, optimized for some given architecture, into a high-level library has been proposed by Brewer [Bre95] and may be of interest in this case. Providing a dedicated library to implement load-balancing is also proposed by George [Geo99], who addresses the problem of dynamic load-balancing via array distribution for the class of iterative algorithms with a SPMD implementation. He introduces the `DParLib`, a library that allows the programmer to produce statistics about load-balance during the execution. The information can then be used to redistribute arrays from one iteration to the other. However, the library does not take into account possible communication-computation overlaps that we need in our second test application.

6 Conclusion

We have discussed in this paper some program transformation strategies for Grid applications, aiming at load-balancing the resources workloads. From the observation of some real application source codes, we have described different situations in which adapted program transformations can greatly influence the application performances. It appears that the source code must be finely analyzed to choose which load-balancing solution best matches the problem. In the first example, we have put forward influent parameters for static load-balancing. In the second example we have shown that a possible transformation strategy to overcome the constraints on data-distribution and keep a good communication-computation overlap can be the process emulation technique.
Future work should be done in several directions. We first need to further investigate the communication schemes used in real applications and how they perform in a Grid environment. A classification of the communication types may be used by a software tool to select appropriate transformation strategies. We also believe the programmer should interact with the tool to guide program transformations as he can bring useful information about the application. Work in progress concerns the development of a user-level library containing load-balanced communication patterns, such as scatter and master/slave communication schemes. Subsequent work should allow the programmer to insert directives in its code. A transformation tool could then interpret the directives and insert the appropriate library calls in the code.

References

[BCF$^+$01] Olivier Beaumont, Larry Carter, Jeanne Ferrante, Arnaud Legrand, and Yves Robert. Bandwidth-centric allocation of independent tasks on heterogeneous platforms. Technical Report 4210, INRIA, Rhône-Alpes, June 2001.

[BDRV99] Pierre Boulet, Jack Dongarra, Yves Robert, and Frédéric Vivien. Static tiling for heterogeneous computing platforms. *Parallel Computing*, 25(5):547–568, 1999.

[BGW93] Amnon Barak, Shai Guday, and Richard Wheeler. *The MOSIX Distributed Operating System, Load Balancing for UNIX*, volume 672 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993. `http://www.mosix.cs.huji.ac.il/`.

[Bre95] Eric A. Brewer. High-level optimization via automated statistical modeling. In *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[BWF$^+$96] Fran Berman, Richard Wolski, Silvia Figueira, Jennifer Schopf, and Gary Shao. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of SuperComputing '96*, 1996.

[FK97] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 1997.

[FK98] Ian Foster and Nicholas Karonis. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. *Supercomputing*, November 1998.

[FSB01] Francis Filbet, Eric Sonnendrücker, and Pierre Bertrand. Conservative numerical schemes for the Vlasov equation. *Journal of Comput. Phys.*, 172:166–187, 2001.

[Geo99] William George. Dynamic load-balancing for data-parallel MPI programs. In *Message Passing Interface developers and users conference*, pages 95–100, March 1999.

[GGM02] Marc Grunberg, Stéphane Genaud, and Catherine Mongenet. Parallel seismic ray-tracing in a global earth mesh. In *Proceedings of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, June 2002.

[KK92] Orly Kremien and Jeff Kramer. Methodical analysis of adaptative load sharing algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 3:747–760, 1992.

[SWB98] Gary Shao, Rich Wolski, and Fran Berman. Performance effects of scheduling strategies for master/slave distributed applications. Technical Report CS98-598, UCSD CSE Dept., University of California, San Diego, September 1998.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

[VF02] Eric Violard and Francis Filbet. Parallelization of a Vlasov solver by communication overlapping. In *Proceedings of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, June 2002.

[WSH99] Rich Wolski, Neil Spring, and Jim Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6):757–768, October 1999.

A Our test Grid

We give here some details on the Grid testbed in which our experiments occured. The testbed is based on the Globus toolkit [FK97] (version 1.1.4) and MPICH-G2 [FK98] (version 1.2.2.3 with our home-made patch to make it really cross-platforms).

site	*resources*	*#procs*	*proc type*
Illkirch 1	PCs	6	Intel
Illkirch 2	Origin 2000	52	Mips R10k+R14k
Strasbourg	Sun	16	Ultra-Sparc
Clermont	PCs	4	Intel
Montpellier	Origin 3800	512	Mips R14k

(a) Available resources

processor	*freq*	*rating*
AMD Athlon	1400	1.61
Intel PIII	933	1.13
Intel PIII	800	1.00
MIPS R14k	500	1.05
MIPS R12k	300	0.65
MIPS R10k	195	0.46

(b) Speed ratings

Table 3: The test Grid.

The resources composing the testbed are shown in table 3(a). Table 3(b) gives some of the ratings we assigned to processors. This ratings come from series of benchmarks we performed on our applications[8]. The great heterogeneity of processors is noticeable.

Bandwidth (MB/s)						Latency (s)				
Illkirch 1	*Illkirch 2*	*Strasbourg*	*Clermont*	*Montpellier*		*Montpellier*	*Clermont*	*Strasbourg*	*Illkirch 2*	*Illkirch 1*
7.5	4	3.7	0.6	0.5	*Illkirch 1*	0.67	0.18	0.02	0.02	0.01
	30	2.5	0.6	0.2	*Illkirch 2*	0.74	0.29	0.02	0.002	
		22	0.6	0.2	*Strasbourg*	0.74	0.29	0.02		
				0.26	*Clermont*	0.57				
				35	*Montpellier*	0.002				

Table 4: Average Bandwidths and Latencies on the test Grid.

Table 4 shows the network links between the different sites during the period when the experiments took place. This values have been measured either by running a ping application or by interrogation of a NWS [WSH99] between two sites.

B Discussion on the scatter load-balancing

We detail in this appendix the developments that lead to results of section 3.3. We first discuss a known algorithm that can be used to solve equation (1) in appendix B.1. Then, we show that equation (2) is a shortcut to solve this problem for values d'_i in $\mathbb{R}$. From the solutions d'_i, the previous algorithm allows to find optimal values d_i in $\mathbb{N}$ (the algorithm is our *round* function) and hence the assertion at the end of section 3.3.
We then infered a generic expression for the case of increasing linear functions.

[8]On IRIX and Sun OS we used vendor supplied compilers and gcc on Linux. We arbitrarily chose a rating of 1 for the Intel 800MHz.

B.1 Minimizing algorithm

The algorithm outlined here has been exposed in [BDRV99] where it is fully explained. We only exhibit the principles of the algorithm in the following.

Given a natural D and a set of increasing function g_i, we call best P-uples $d_1, \ldots, d_P$ those minimizing $max\{g_i(d_i) \mid i \in [1, P]\}$. The main idea is to recursively construct a best P-uple $d_1, \ldots, d_P$ such that $\sum_{i=1}^{P} d_i = D$ from a best P-uple $d'_1, \ldots, d'_P$ such that $\sum_{i=1}^{P} d'_i = D - 1$. The algorithm is :

when $D = 0$, $d_i = 0 \ \forall i$

when $D > 0$, let $(d'_i)_i$ be a solution for $(D - 1)$. We search an index i_0 such that $g_i(d'_i + 1)$ is minimal. Then we define a solution $(d_i)_i$ for D by $d_i = d'_i \ \forall \ i \neq i_0$, $d_{i_0} = d'_{i_0} + 1$.

B.2 Shortcut to the algorithm

We try here to reduce the number of operations involved in the calculation of a minimizing solution. (The complexity of the Minimizing algorithm is $P \times D$ operations.) The main idea is to find an explicit formula for a solution in $\mathbb{R}$ and to use the Minimizing algorithm to find a solution in $\mathbb{N}$. For this purpose we must start by some remarks.

Let's note $(g_i)_{i=1,\ldots P}$ a family of increasing functions. We remark that if we can find a P-uple $d_1, \ldots, d_P$ such that $\sum_{i=1}^{P} d_i = D$ and for which $g_1(d_1) = \ldots = g_P(d_P)$ then this P-uple minimizes $max\{g_i(d_i) \mid i \in [1, P]$ and $\sum_{i=1}^{P} d_i = D\}$.

Proof. Consider (d_i) a P-uple such as described above, and (d'_i) another P-uple with sum D. Because $(d_i) \neq (d'_i)$ and both sums are equals to D there is an index i_0 such that $d'_{i_0} > d_{i_0}$, which implies that $g_{i_0}(d'_{i_0}) \geq g_{i_0}(d_{i_0})$ since g_{i_0} is an increasing function. By assumptions $g_{i_0}(d_{i_0}) = g_1(d_1) = \ldots = g_P(d_P) = max\{g_i(d_i) \mid i \in [1, P]\}$, therefore $g_{i_0}(d'_{i_0}) \geq max\{g_i(d_i) \mid i \in [1, P]\}$ and then $max\{g_i(d'_i) \mid i \in [1, P]\} \geq max\{g_i(d_i) \mid i \in [1, P]\}$.

□

Note that this result is true whether d_i are naturals or reals.

If no such P-uple can be found in $\mathbb{N}$ but one can be found in $\mathbb{R}$ (let's write it (d_i)), then the P-uple (d'_i) built on the floor-rounded values of (d_i) (e.g. $d'_i = \lfloor d_i \rfloor$) minimizes $max\{g_i(\delta_i) \mid i \in [1, P]$ and $\sum_{i=1}^{P} \delta_i = D'\}$, where $D' = \sum_{i=1}^{P} d'_i$.

Proof. Consider (δ_i) another P-uple with sum D'. There is an index i_0 such that $\delta_{i_0} > d'_{i_0}$. Beeing naturals this inequation can be expressed $\delta_{i_0} \geq d'_{i_0} + 1$, which implies that $g_{i_0}(\delta_{i_0}) \geq g_{i_0}(d'_{i_0} + 1) \geq g_{i_0}(d_{i_0})$. By assumption $g_{i_0}(d_{i_0}) = max\{g_i(d_i) \mid i \in [1, P]\}$, thus $g_{i_0}(\delta_{i_0}) \geq max\{g_i(d_i) \mid i \in [1, P]\}$ and then $max\{g_i(\delta_i) \mid i \in [1, P]\} \geq max\{g_i(d_i) \mid i \in [1, P]\}$. Obviously $max\{g_i(d'_i) \mid i \in [1, P]\} \leq max\{g_i(d_i) \mid i \in [1, P]\}$, which leads to the result.

□

Starting from this natural solution with sum D', the Minimizing algorithm allows us to build a natural P-uple with sum D which minimizes $max\{g_i(d_i) \mid i \in [1, P],\ d_i \in \mathbb{N}\ , \sum_{i=1}^{P} d_i = D\}$.

In our work assumptions $comp_i$ and $comm_i$ are increasing functions, thus $g_i = comp_i + comm_i$ are also increasing functions, which allow us to use the results exposed in this section.

B.3 Refinement for linear computations

In section 3.3 we found a general expression for the d_i. Though we clearly pointed out the fact that these expressions were correct we can show it more formaly by considering the matrix expression of system (2):

$$\begin{pmatrix} c_1 & -c_2 & 0 & 0 & \dots & 0 \\ c_1 & 0 & -c_3 & 0 & \dots & 0 \\ c_1 & 0 & 0 & -c_4 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ c_1 & 0 & 0 & 0 & \dots & -c_P \\ 1 & 1 & 1 & 1 & \dots & 1 \end{pmatrix} \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{P-1} \\ d_P \end{pmatrix} = \begin{pmatrix} l_2 - l_1 \\ l_3 - l_1 \\ l_4 - l_1 \\ \vdots \\ l_P - l_1 \\ D \end{pmatrix}$$

By recurence it is easy to show that the determinant of the matrix is

$$\sum_{j=1}^{P} \left(\frac{\prod_{k=1}^{P} c_k}{c_j} \right)$$

which implies that there is one solution $(d_i)_i$ and that it is unique. Anyway, that does not involve that this solution does not have negative elements.

When one of the calculated d_i is negative, that only means that there is no satisfying solution to equation (2). In order to find a minimalizing solution in such a case one has to go back to Minimizing algorithm.