Load-Balancing Scatter Operations for Grid Computing*

Stéphane Genaud[1] Arnaud Giersch[1] Frédéric Vivien[2]

[1] ICPS-LSIIT - UMR 7005
Université Louis Pasteur, Strasbourg
{genaud,giersch}@icps.u-strasbg.fr

[2] LIP, École normale supérieure de Lyon
INRIA
Frederic.Vivien@ens-lyon.fr

Abstract

We present solutions to statically load-balance scatter operations in parallel codes run on Grids. Our load-balancing strategy is based on the modification of the data distributions used in scatter operations. We need to modify the user source code, but we want to keep the code as close as possible to the original. Hence, we study the replacement of scatter operations with a parameterized scatter, allowing a custom distribution of data. The paper presents: 1) a general algorithm which finds an optimal distribution of data across processors; 2) a quicker guaranteed heuristic relying on hypotheses on communications and computations; 3) a policy on the ordering of the processors. Experimental results with an MPI scientific code of seismic tomography illustrate the benefits obtained from our load-balancing.

1. Introduction

Traditionally, users have developed scientific applications with a parallel computer in mind, assuming an homogeneous set of processors linked with an homogeneous and fast network. However, *Grids* [10] of computational resources usually include heterogeneous processors, and heterogeneous network links that are orders of magnitude slower than in a parallel computer. Therefore, the execution on Grids of applications designed for parallel computers usually leads to poor performance as the distribution of workload does not take the heterogeneity into account. Hence the need for tools able to analyze and transform existing parallel applications to improve their performances on heterogeneous environments by load-balancing their execution. Furthermore, we are not willing to fully rewrite the original applications but we are rather seeking transformations which modify the original source code as little as possible.

*This research is supported by the French Ministry of Research through the ACI-GRID program.

Among the usual operations found in parallel codes is the *scatter* operation, which is one of the *collective* operations usually shipped with message passing libraries. For instance, the mostly used message passing library MPI [16] provides a `MPI_Scatter` primitive that allows the programmer to distribute even parts of data to the processors in the MPI communicator.

The less intrusive modification enabling a performance gain in an heterogeneous environment consists in using a communication library adapted to heterogeneity. Thus, much work has been devoted to that purpose: for MPI, numerous projects including Magpie [15], MPI-StarT [13], and MPICH-G2 [8], aim at improving communications performance in presence of heterogeneous networks. Most of the gain is obtained by reworking the design of collective communication primitives. For instance, MPICH-G2 performs often better than MPICH to disseminate information held by a processor to several others. While MPICH always use a binomial tree to propagate data, MPICH-G2 is able to switch to a flat tree broadcast when network latency is high [14]. Making the communication library aware of the precise network topology is not easy: MPICH-G2 queries the underlying Globus [9] environment to retrieve information about the network topology that the user may have specified through environment variables. Such network-aware libraries bring interesting results as compared to standard communication libraries. However, these improvements are often not sufficient to attain performance considered acceptable by users when the processors are also heterogeneous. Balancing the computation tasks over processors is also needed to take benefit from Grids.

The typical usage of the scatter operation is to spawn an SPMD computation section on the processors after they received their piece of data. Thereby, if the computation load on processors depends on the data received, we can use the scatter operation as a means to load-balance computations, provided the items in the data set to scatter are independent. MPI provides the primitive `MPI_Scatterv` that allows to distribute *unequal* shares of data. We claim that replacing `MPI_Scatter` by `MPI_Scatterv` calls parameterized with clever distributions may lead to great performance improvements at low cost. In term of source code rewriting, the transformation of such operations does not require a deep source code re-organization, and it can easily be automated in a software tool. Our problem is thus to load-balance the execution by computing a data distribution depending on the processors speeds and network links bandwidths.

In Section 2 we present our target application, a real scientific application in geophysics, written in MPI, that we ran to ray-trace the full set of seismic events of year 1999. In Section 3 we present our load-balancing techniques, in Section 4 the processor ordering policy we derive from a case study, in Section 5 our experimental results, in Section 6 the related works, and we conclude in Section 7.

2. Motivating example

2.1. Seismic tomography

The geophysical code we consider is in the seismic tomography field. The general objective of such applications is to build a global seismic velocity model of the Earth interior. The various velocities found at the different points discretized by the model (generally a mesh) reflect the physical rock properties in those locations. The seismic waves velocities are computed from the seismograms recorded by captors located all around the globe: once analyzed, the wave type, the earthquake hypocenter and the captor locations as well as the wave travel time are determined.

From these data, a tomography application reconstructs the event using an initial velocity model. The wave propagation from the source hypocenter to a given captor defines a path, that the application evaluates given properties of the initial velocity model. The time for the wave to propagate along this evaluated path is then compared to the actual travel time, and in a final step, a new velocity model that minimizes those differences is computed. This process is more accurate if the new model better fits numerous such paths in many locations inside the Earth, and is therefore very computationally demanding.

2.2. The example application

We now outline how the application under study exploits the potential parallelism of the computations, and how the tasks are distributed across processors. Recall that the input data is a set of seismic waves characteristics each described by a pair of 3D coordinates (the coordinates of the earthquake source and those of the receiving captor) plus the wave type. With these characteristics, a seismic wave can be modeled by a set of *ray paths* that represents the wavefront propagation. Seismic wave characteristics are sufficient to perform the ray-tracing of the whole associated ray path. Therefore, all ray paths can be traced independently. The existing parallelization of the application (presented in [12]) assumes an homogeneous set of processors (the implicit target being 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 processors involved, and n the number of data items. The `MPI_Scatter` instruction is executed by the root and the computation processors. The processor identified as `ROOT` performs a send of contiguous blocks of $\lfloor n/P \rfloor$ elements from the `raydata` buffer to all processors of the group while all processors make a receive operation of their respective data in the `rbuff` buffer. For sake of simplicity the remaining $(n \bmod P)$ items distribution is not shown here. Figure 1 shows a potential execution of this communication operation, with P_4 as root process.

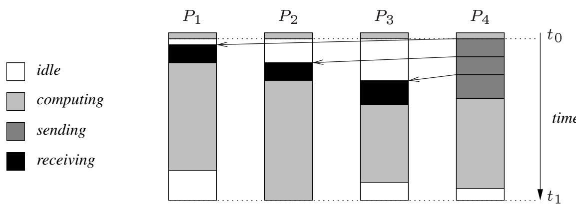

Figure 1. A scatter communication followed by a computation phase.

2.3. Hardware model

Figure 1 outlines the behavior of the scatter operation as it was observed during the applications runs on our test Grid (described in Section 5.1). This behavior is an indication on the networking capabilities of the root node: it can send to at most one destination node at a time. This is the single-port model of [4] which is realistic for Grids as many nodes are simple PCs with full-duplex network cards. As the root processor sends data to processors in turn[1] a receiving processor actually begins its communication after all previous

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

processors have been served. This leads to a "stair effect" represented on Figure 1 by the end times of the receive operations (black boxes).

3. Static load-balancing

As the overall execution time after load-balancing is rather small, we make the assumption that the grid characteristics do not change during the computation and we only consider static load-balancing. Note also that the computed distribution is not necessarily based on static parameters estimated for the whole execution: a monitor daemon process (like [19]) running aside the application could be queried just before a scatter operation to retrieve the instantaneous Grid characteristics.

3.1. Framework

We consider a set of p processors: $P_1, \ldots, P_p$, each of them being characterized by 1) the time $T_{\mathrm{comp}}(i,x)$ it takes to compute x data items; 2) the time $T_{\mathrm{comm}}(i,x)$ it takes to receive x data items from the root process. We want to process n data items. We look for a distribution $n_1, \ldots, n_p$ of these data over the p processors that minimizes the overall computation time. In all this paper the root processor will be the last process, P_p, as it can only start to process its share of the data items *after* it has sent the other data items to the other processors. As the root processor sends data to processors in turn, processor P_i begins its communication after processors $P_1, \ldots, P_{i-1}$ have been served, which takes a time $\sum_{j=1}^{i-1} T_{\mathrm{comm}}(j, n_j)$. The root takes $T_{\mathrm{comm}}(i, n_i)$ to send to P_i its data, and P_i takes $T_{\mathrm{comp}}(i, n_i)$ to process them. Thus, P_i ends its processing at time:

$$T_i = \sum_{j=1}^{i} T_{\mathrm{comm}}(j, n_j) + T_{\mathrm{comp}}(i, n_i). \tag{1}$$

The time, T, taken by our system to compute the set of n data items is therefore:

$$\begin{aligned} T &= \max_{1 \le i \le p} T_i \\ &= \max_{1 \le i \le p} \left(\sum_{j=1}^{i} T_{\mathrm{comm}}(j, n_j) + T_{\mathrm{comp}}(i, n_i) \right), \end{aligned} \tag{2}$$

and we are looking for the distribution $n_1, \ldots, n_p$ minimizing this duration.

3.2. An exact solution by dynamic programming

Studying Equation (2) we remark that the time to process n data on processors 1 to p is equal to the maximum of 1) the time taken by the root to send n_1 data to P_1 plus the time taken by P_1 to process them; 2) the time for processors 2 to p to process $n - n_1$ data *plus* the time for the root to send the n_1 data to P_1. This leads to the dynamic programming Algorithm 1 (the distribution is expressed as a list, hence the use of the list constructor cons). In Algorithm 1, $\mathrm{cost}[d,i]$ denotes the cost of the processing of d data items over the processors P_i through P_p. $\mathrm{solution}[d,i]$ is a list describing the distribution of d data items over the processors P_i through P_p to achieve the minimal execution time $\mathrm{cost}[d,i]$.

Algorithm 1 Compute an optimal distribution of n data over p processors

```
cost[0, p] ← 0
solution[0, p] ← cons(0, NIL)
for d ← 1 to n do
  cost[d, p] ← T_comm(p, d) + T_comp(p, d)
  solution[d, p] ← cons(d, NIL)
for i ← p − 1 to 1 do
  cost[0, i] ← 0
  solution[0, i] ← cons(0, solution[0, i + 1])
  for d ← 1 to n do
    min ← cost[d, i + 1]
    sol ← 0
    for e ← 1 to d do
      m ← max(T_comm(i, e) + T_comp(i, e),
                        T_comm(i, e) + cost[d − e, i + 1])
      if m < min then
        min ← m
        sol ← e
    cost[d, i] ← min
    solution[d, i] ← cons(sol, solution[d − sol, i + 1])
return (cost[n, 1], solution[n, 1])
```

Algorithm 1 has a complexity of $O(p \cdot n^2)$, which may be prohibitive. But Algorithm 1 only assumes that the functions $T_{\mathrm{comm}}(i,x)$ and $T_{\mathrm{comp}}(i,x)$ are non-negative. We now present a more efficient heuristic valid for simple cases.

3.3. A guaranteed heuristic using linear programming

In this section, we make the hypothesis that all the functions $T_{\mathrm{comm}}(i,n)$ and $T_{\mathrm{comp}}(i,n)$ are affine in n, increasing, and non-negative (for $n \ge 0$). Equation (2) can then be coded into the following linear program:

$$\begin{cases} \text{Minimize } T \text{ such that} \\ \forall i \in [1,p],\ n_i \ge 0, \\ \sum_{i=1}^{p} n_i = n, \\ \forall i \in [1,p],\ T \ge \sum_{j=1}^{i} T_{\mathrm{comm}}(j, n_j) + T_{\mathrm{comp}}(i, n_i). \end{cases} \tag{3}$$

This linear program must be solved in integer to find an integer solution. However, we can solve it in rational to obtain an optimal *rational* solution $n_1, \ldots, n_p$ that we round up to obtain an integer solution $n'_1, \ldots, n'_p$ with $\sum_i n'_i = n$. Let T' be the execution time of this solution, T be the time of the rational solution, and T_{opt} the time of the optimal integer solution. If $|n_i - n'_i| \leq 1$ for any i, which is easily enforced by the rounding scheme described below, then:

$$T_{\text{opt}} \leq T' \leq T_{\text{opt}} + \sum_{j=1}^{p} T_{\text{comm}}(j, 1) + \max_{1 \leq i \leq p} T_{\text{comp}}(i, 1). \tag{4}$$

Indeed,

$$T' = \max_{1 \leq i \leq p} \left(\sum_{j=1}^{i} T_{\text{comm}}(j, n'_j) + T_{\text{comp}}(i, n'_i) \right). \tag{5}$$

By hypothesis, $T_{\text{comm}}(j, x)$ and $T_{\text{comp}}(j, x)$ are non-negative, increasing, and affine functions. Therefore,

$$\begin{aligned} T_{\text{comm}}(j, n'_j) &= T_{\text{comm}}(j, n_j + (n'_j - n_j)) \\ &\leq T_{\text{comm}}(j, n_j + |n'_j - n_j|) \\ &\leq T_{\text{comm}}(j, n_j) + T_{\text{comm}}(j, |n'_j - n_j|) \\ &\leq T_{\text{comm}}(j, n_j) + T_{\text{comm}}(j, 1) \end{aligned}$$

and we have an equivalent upper bound for $T_{\text{comp}}(j, n'_j)$. Using these upper bounds to over-approximate the expression of T' given by Equation (5) we obtain:

$$T' \leq \max_{1 \leq i \leq p} \left(\sum_{j=1}^{i} \left(T_{\text{comm}}(j, n_j) + T_{\text{comm}}(j, 1) \right) \right. \\ \left. + T_{\text{comp}}(i, n_i) + T_{\text{comp}}(i, 1) \right) \tag{6}$$

which implies Equation (4) knowing that $T_{opt} \leq T'$, $T = \max_{1 \leq i \leq p}(\sum_{j=1}^{i} T_{\text{comm}}(j, n_j) + T_{\text{comp}}(i, n_i))$, and $T \leq T_{opt}$.

Rounding scheme. Our rounding scheme is trivial: first we round, to the nearest integer, the n_i which is nearest to an integer. Doing so we obtain n'_i and we make an approximation error of $e = n'_i - n_i$ (with $|e| < 1$). If e is negative (resp. positive), n_i was underestimated (resp. overestimated) by the approximation. Then we round to its ceiling (resp. floor), one of the remaining n_js which is the nearest to its ceiling $\lceil n_j \rceil$ (resp. floor $\lfloor n_j \rfloor$), we obtain a new approximation error of $e = e + n'_j - n_j$ (with $|e| < 1$), and so on until there only remains to approximate only one of the n_is, say n_k. Then we let $n'_k = n_k + e$. The distribution n'_1, $\ldots$, n'_p is thus integer, $\sum_{1 \leq i \leq p} n'_i = d$, and each n'_i differs from n_i by less than one.

3.4. Choice of the root process

We make the assumption that, originally, the n data items that must be processed are stored on a single computer, denoted $\mathcal{C}$. A processor of $\mathcal{C}$ may or may not be used as the root processor. If the root processor is not on $\mathcal{C}$, then the whole execution time is equal to the time needed to transfer the data from $\mathcal{C}$ to the root processor, plus the execution time as computed by Algorithm 1. The best root processor is then the processor minimizing this whole execution time, when picked as root. This is just the result of a minimization over the p candidates.

4. A case study: solving in rational with linear communication and computation times

In this section we study a simple and theoretical case. We make the hypothesis that all the functions $T_{\text{comm}}(i, n)$ and $T_{\text{comp}}(i, n)$ are linear in n. In other words, there are constants λ_i and μ_i such that $T_{\text{comm}}(i, n) = \lambda_i \cdot n$ and $T_{\text{comp}}(i, n) = \mu_i \cdot n$.

Also, we only look for a rational solution and not an integer one as we should. This case study will enable us to define a policy on the order in which the processors must receive their data. Indeed, in our simple case the processor ordering leading to the shortest execution time is quite simple as we show in Section 4.3. Before that we prove in Section 4.2 that there always is an optimal (rational) solution in which all the working processors have the same ending time. We also show the condition for a processor to receive a share of the whole work. As this condition comes from the expression of the execution duration when all processors have to process a share of the whole work and finishes at the same date, we begin by studying this case in Section 4.1. Finally, in Section 4.4, we derive from our case study some consequences for the general case.

4.1. Execution duration

Theorem 1 (Execution duration) *If we are looking for a rational solution, if each processor P_i receives a (non empty) share n_i of the whole set of n data items and if all processors end their computation at a same date t, then the execution time is*

$$t = \frac{n}{\sum_{i=1}^{p} \frac{1}{\lambda_i + \mu_i} \cdot \prod_{j=1}^{i-1} \frac{\mu_j}{\lambda_j + \mu_j}} \tag{7}$$

and the processor P_i receives

$$n_i = \frac{1}{\lambda_i + \mu_i} \cdot \left(\prod_{j=1}^{i-1} \frac{\mu_j}{\lambda_j + \mu_j} \right) \cdot t \tag{8}$$

data to process.

Proof We want to express t and n_i as functions of n. Equation (2) states that processor P_i ends its processing at time: $T_i = \sum_{j=1}^{i} T_{\text{comm}}(j, n_j) + T_{\text{comp}}(i, n_i)$. So, with our current hypotheses: $T_i = \sum_{j=1}^{i} \lambda_j \cdot n_j + \mu_i \cdot n_i$. Thus, $n_1 = \frac{t}{\lambda_1 + \mu_1}$ and, for $i \in [2, p]$,

$$T_i = T_{i-1} - \mu_{i-1} \cdot n_{i-1} + (\lambda_i + \mu_i) \cdot n_i.$$

As by hypothesis all processors end their processing at the same time, $T_i = T_{i-1} = t$, $n_i = \frac{\mu_{i-1}}{\lambda_i + \mu_i} \cdot n_{i-1}$, and we find Equation (8).

To express the execution duration t as a function of n we just sum Equation (8) for all values of i in $[1, p]$:

$$n = \sum_{i=1}^{p} n_i = \sum_{i=1}^{p} \frac{1}{\lambda_i + \mu_i} \cdot \left(\prod_{j=1}^{i-1} \frac{\mu_j}{\lambda_j + \mu_j} \right) \cdot t$$

which is equivalent to Equation (7). ∎

In the rest of this paper we note:

$$D(P_1, \ldots, P_p) = \frac{1}{\sum_{i=1}^{p} \frac{1}{\lambda_i + \mu_i} \cdot \prod_{j=1}^{i-1} \frac{\mu_j}{\lambda_j + \mu_j}}.$$

and so we have $t = n \cdot D(P_1, \ldots, P_p)$ under the hypotheses of Theorem 1.

4.2. Simultaneous endings

In this paragraph we exhibit a condition on the costs functions $T_{\text{comm}}(i, n)$ and $T_{\text{comp}}(i, n)$ of a set of processors that is necessary and sufficient to have an optimal rational solution where each processor receives a non-empty share of data, and all processors end at the same date. This tells us when Theorem 1 can be used to find a rational solution to our system.

Theorem 2 (Simultaneous endings) *Given P processors, $P_1, \ldots, P_i, \ldots, P_p$, whose communication and computation duration functions $T_{\text{comm}}(i, n)$ and $T_{\text{comp}}(i, n)$ are linear in n, there exists an optimal rational solution where each processor receives a non-empty share of the whole set of data, and all processors end their computation at the same date, if and only if*

$$\forall i \in [1, p-1], \quad \lambda_i \leq D(P_{i+1}, \ldots, P_p)$$

Proof The proof is made by induction on the number of processors. If there is only one processor, then the theorem is trivially true. We shall next prove that if the theorem is true for p processors, then it is also true for $p+1$ processors.

Suppose we have $p+1$ processors $P_1, \ldots, P_{p+1}$. An optimal solution for $P_1, \ldots, P_{p+1}$ to compute n data items is obtained by giving $\alpha \cdot n$ items to P_1 and $(1 - \alpha) \cdot n$ items to $P_2, \ldots, P_{p+1}$ with α in $[0, 1]$. The end date for the processor P_1 is then $t_1(\alpha) = (\lambda_1 + \mu_1) \cdot n \cdot \alpha$.

As the theorem is supposed to be true for p processors, we know that there exists an optimal rational solution where the processors P_2 to P_{p+1} all work and finish their work simultaneously, if and only if $\forall i \in [2, p]$, $\lambda_i \leq D(P_{i+1}, \ldots, P_{p+1})$. In this case, by Theorem 1, the time taken by $P_2, \ldots, P_{p+1}$ to compute $(1 - \alpha) \cdot n$ data is $(1 - \alpha) \cdot n \cdot D(P_2, \ldots, P_{p+1})$. So, the processors $P_2, \ldots, P_{p+1}$ all end at the same date $t_2(\alpha) = \lambda_1 \cdot n \cdot \alpha + k \cdot n \cdot (1 - \alpha) = k \cdot n + (\lambda_1 - k) \cdot n \cdot \alpha$ with $k = D(P_2, \ldots, P_{p+1})$.

If $\lambda_1 \leq k$, then $t_1(\alpha)$ is strictly increasing, and $t_2(\alpha)$ is decreasing. Moreover, we have $t_1(0) < t_2(0)$ and $t_1(1) > t_2(1)$, thus the whole end date $\max(t_1(\alpha), t_2(\alpha))$ is minimized for an unique α in $]0, 1[$, when $t_1(\alpha) = t_2(\alpha)$. In this case, each processor has some data to compute and they all end at the same date.

On the contrary, if $\lambda_1 > k$, then $t_1(\alpha)$ and $t_2(\alpha)$ are both strictly increasing, thus the whole end date $\max(t_1(\alpha), t_2(\alpha))$ is minimized for $\alpha = 0$. In this case, processor P_1 has nothing to compute and its end date is 0, while processors P_2 to P_{p+1} all end at a same date $k \cdot n$.

Thus, there exists an optimal rational solution where each of the $p+1$ processors $P_1, \ldots, P_{p+1}$ receives a non-empty share of the whole set of data, and all processors end their computation at the same date, if and only if $\forall i \in [1, p]$, $\lambda_i \leq D(P_{i+1}, \ldots, P_{p+1})$. ∎

The proof of Theorem 2 shows that any processor P_i such that $\lambda_i > D(P_{i+1}, \ldots, P_p)$ is not interesting for our problem: using it will only increase the whole processing time. Therefore, we just forget those processors and Theorem 2 states that there is an optimal rational solution where the remaining processors are all working and have the same end date.

4.3. Processor ordering policy

As we have stated in Section 2.3, the root processor sends data to processors in turn and a receiving processor actually begins its communication after all previous processors have received their shares of data. Moreover, in the MPICH implementation of MPI, the order of the destination processors in scatter operations follows the processors ranks defined by the program(mer). Therefore, setting the processor ranks influence the order in which the processors start to receive and process their share of the whole work. Equation (7) shows that in our case the overall computation time is not symmetric in the processors but depends on their ordering. Therefore we must carefully defines this ordering in order to speed-up the whole computation. It appears that in our current case, the best ordering is quite simple:

Theorem 3 (Processor ordering policy) *When all the functions $T_{\text{comm}}(i, n)$ and $T_{\text{comp}}(i, n)$ are linear in n,* *when for any i in $[1, p-1]$, $\lambda_i \leq D(P_{i+1}, \ldots, P_p)$, and when we are only looking for a rational solution, then the smallest execution time is achieved when the processors (the root processor excepted) are ordered in decreasing order of their bandwidth (from P_1, the processor connected to the root process with the highest bandwidth, to P_{p-1}, the processor connected to the root processor with the smallest bandwidth), the last processor being the root processor.*

Proof We consider any ordering $P_1, \ldots, P_p$, of the processors, except that P_p is the root processor (as we have explained in Section 3.1). We consider any permutation π of such an ordering. In other words, we consider any order $P_{\pi(1)}, \ldots, P_{\pi(p)}$ of the processors such that there exists $k \in [1, p-2]$, $\pi(k) = k+1$, $\pi(k+1) = k$, and $\forall j \in [1,p] \setminus \{k, k+1\}, \pi(j) = j$ (note that $\pi(p) = p$).

We denote by t_π (resp. t) the best (rational) execution time when the processors are ordered $P_{\pi(1)}, \ldots, P_{\pi(p)}$ (resp. $P_1, \ldots, P_p$). We must show that if P_{k+1} is connected to the root processor with an higher bandwidth than P_k, then t_π is strictly smaller than t. In other words we must show the implication:

$$\lambda_{k+1} < \lambda_k \quad \Rightarrow \quad t_\pi < t. \tag{9}$$

Therefore, we study the sign of $t_\pi - t$.

In this difference, we can replace t by its expression as stated by Equation (7) as, by hypothesis, for any i in $[1, p-1]$, $\lambda_i \leq D(P_{i+1}, \ldots, P_p)$. For t_π, things are a bit more complicated. If, for any i in $[1, p-1]$, $\lambda_{\pi(i)} \leq D(P_{\pi(i+1)}, \ldots, P_{\pi(p)})$, Theorems 2 and 1 apply:

$$t_\pi = \frac{n}{\sum_{i=1}^{p} \frac{1}{\lambda_{\pi(i)} + \mu_{\pi(i)}} \cdot \prod_{j=1}^{i-1} \frac{\mu_{\pi(j)}}{\lambda_{\pi(j)} + \mu_{\pi(j)}}}. \tag{10}$$

On the opposite, if there exists a value i in $[1, p-1]$ such that $\lambda_{\pi(i)} > D(P_{\pi(i+1)}, \ldots, P_{\pi(p)})$, then Theorem 2 states that the optimal execution time cannot be achieved on a solution where each processor receives a non-empty share of the whole set of data and all processors end their computation at the same date. Therefore, any solution where each processor receives a non-empty share of the whole set of data and all processors end their computation at the same date leads to an execution time strictly greater than t_π and:

$$t_\pi < \frac{n}{\sum_{i=1}^{p} \frac{1}{\lambda_{\pi(i)} + \mu_{\pi(i)}} \cdot \prod_{j=1}^{i-1} \frac{\mu_{\pi(j)}}{\lambda_{\pi(j)} + \mu_{\pi(j)}}}. \tag{11}$$

Equations (10) and (11) are summarized by:

$$t_\pi \leq \frac{n}{\sum_{i=1}^{p} \frac{1}{\lambda_{\pi(i)} + \mu_{\pi(i)}} \cdot \prod_{j=1}^{i-1} \frac{\mu_{\pi(j)}}{\lambda_{\pi(j)} + \mu_{\pi(j)}}} \tag{12}$$

and proving the following implication:

$$\lambda_{k+1} < \lambda_k \Rightarrow \frac{n}{\sum_{i=1}^{p} \frac{1}{\lambda_{\pi(i)} + \mu_{\pi(i)}} \cdot \prod_{j=1}^{i-1} \frac{\mu_{\pi(j)}}{\lambda_{\pi(j)} + \mu_{\pi(j)}}} < t \tag{13}$$

will prove Equation (9). Hence, we study the sign of

$$\sigma = \frac{n}{\sum_{i=1}^{p} \frac{1}{\lambda_{\pi(i)} + \mu_{\pi(i)}} \cdot \prod_{j=1}^{i-1} \frac{\mu_{\pi(j)}}{\lambda_{\pi(j)} + \mu_{\pi(j)}}} - \frac{n}{\sum_{i=1}^{p} \frac{1}{\lambda_i + \mu_i} \cdot \prod_{j=1}^{i-1} \frac{\mu_j}{\lambda_j + \mu_j}}.$$

As, in the above expression, both denominators are obviously (strictly) positive, the sign of σ is the sign of:

$$\sum_{i=1}^{p} \frac{1}{\lambda_i + \mu_i} \cdot \prod_{j=1}^{i-1} \frac{\mu_j}{\lambda_j + \mu_j} - \sum_{i=1}^{p} \frac{1}{\lambda_{\pi(i)} + \mu_{\pi(i)}} \cdot \prod_{j=1}^{i-1} \frac{\mu_{\pi(j)}}{\lambda_{\pi(j)} + \mu_{\pi(j)}}. \tag{14}$$

We want to simplify the second sum in Equation (14). Thus we remark that for any value of $i \in [1, k] \cup [k+2, p]$ we have:

$$\prod_{j=1}^{i-1} \frac{\mu_{\pi(j)}}{\lambda_{\pi(j)} + \mu_{\pi(j)}} = \prod_{j=1}^{i-1} \frac{\mu_j}{\lambda_j + \mu_j}. \tag{15}$$

In order to take advantage of the simplification proposed by Equation (15), we decompose the second sum in Equation (14) in four terms: the sum from 1 to $k-1$, the terms for k and $k+1$, and then the sum from $k+2$ to p:

$$\begin{aligned} \sum_{i=1}^{p} \frac{1}{\lambda_{\pi(i)} + \mu_{\pi(i)}} \cdot \prod_{j=1}^{i-1} \frac{\mu_{\pi(j)}}{\lambda_{\pi(j)} + \mu_{\pi(j)}} = {} & \sum_{i=1}^{k-1} \frac{1}{\lambda_i + \mu_i} \cdot \prod_{j=1}^{i-1} \frac{\mu_j}{\lambda_j + \mu_j} \\ & + \frac{1}{\lambda_{k+1} + \mu_{k+1}} \cdot \prod_{j=1}^{k-1} \frac{\mu_j}{\lambda_j + \mu_j} \\ & + \frac{1}{\lambda_k + \mu_k} \cdot \frac{\mu_{k+1}}{\lambda_{k+1} + \mu_{k+1}} \cdot \prod_{j=1}^{k-1} \frac{\mu_j}{\lambda_j + \mu_j} \\ & + \sum_{i=k+2}^{p} \frac{1}{\lambda_i + \mu_i} \cdot \prod_{j=1}^{i-1} \frac{\mu_j}{\lambda_j + \mu_j}. \end{aligned} \tag{16}$$

Then we report the result of Equation (16) in Equation (14), we suppress the terms common to both sides of the " $-$ " sign, and we divide the resulting equation by the (strictly) positive term $\prod_{j=1}^{k-1} \frac{\mu_j}{\lambda_j + \mu_j}$. This way, we obtain that σ has the same sign than:

$$\frac{1}{\lambda_k + \mu_k} + \frac{1}{\lambda_{k+1} + \mu_{k+1}} \cdot \frac{\mu_k}{\lambda_k + \mu_k} - \frac{1}{\lambda_{k+1} + \mu_{k+1}} - \frac{1}{\lambda_k + \mu_k} \cdot \frac{\mu_{k+1}}{\lambda_{k+1} + \mu_{k+1}}$$

which is equivalent to:

$$\frac{\lambda_{k+1} - \lambda_k}{(\lambda_k + \mu_k) \cdot (\lambda_{k+1} + \mu_{k+1})}.$$

Therefore, if $\lambda_{k+1} < \lambda_k$, then $\sigma < 0$, Equation (13) holds , and thus Equation (9) also holds.

Therefore, the inversion of processors P_k and P_{k+1} is profitable if the bandwidth from the root processor to processor P_{k+1} was higher than the bandwidth from the root processor to processor P_k. ■

4.4. Consequences for the general case

So, in the general case, how are we going to order our processors? An exact study is feasible even in the general case, if we know the computation and communication characteristics of each of the processors. We can indeed consider all the possible orderings of our p processors, use Algorithm 1 to compute the theoretical execution times, and chose the best result. This is theoretically possible. In practice, for large values of p such an approach is unrealistic. Furthermore, in the general case an analytical study is of course impossible (we cannot analytically handle *any* function $T_{\mathrm{comm}}(i, n)$ or $T_{\mathrm{comp}}(i, n)$).

So, we build from the previous result and we order the processors in decreasing order of the bandwidth they are connected to the root processor with, except for the root processor which is ordered last. Even without the previous study, such a policy should not be surprising. Indeed, the time spent to send its share of the data items to processor P_i is payed by all the processors from P_i to P_p. So the first processor should be the one it is the less expensive to send the data to, and so on. Of course, in practice, things are a bit more complicated as we are working in integers. However, the main idea is roughly the same as we now show.

We only suppose that all the computation and communication functions are linear. Then we denote by:

- T_{opt}^{rat}: the best execution time that can be achieved for a *rational* distribution of the n data items, whatever the ordering for the processors.
- T_{opt}^{int}: the best execution time that can be achieved for an *integer* distribution of the n data items, whatever the ordering for the processors.

Note that T_{opt}^{rat} and T_{opt}^{int} may be achieved on two different ordering of the processors. We take a rational distribution achieving the execution time T_{opt}^{rat}. We round it up to obtain an integer solution, following the rounding scheme described in Section 3.3. This way we obtain an integer distribution of execution time T' with T' satisfying the equation:

$$T' \leq T_{opt}^{rat} + \sum_{j=1}^{p} T_{\mathrm{comm}}(j, 1) + \max_{1 \leq i \leq p} T_{\mathrm{comp}}(i, 1)$$

(the proof being the same than for Equation (4)). However, T' being an integer solution its execution time is obviously at least equal to T_{opt}^{int}. Also, an integer solution being a rational solution, T_{opt}^{int} is at least equal to T_{opt}^{rat}. Hence the bounds:

$$T_{opt}^{int} \leq T' \leq T_{opt}^{int} + \sum_{j=1}^{p} T_{\mathrm{comm}}(j, 1) + \max_{1 \leq i \leq p} T_{\mathrm{comp}}(i, 1)$$

where T' is the execution time of the distribution obtained by rounding up, according to the scheme of Section 3.3, the best rational solution when the processors are ordered in decreasing order of the bandwidth they are connected to the root processor with, except for the root processor which is ordered last.

When all the computation and communication functions are linear our ordering policy is even guaranteed!

5. Experimental results

5.1. Hardware environment

Our experiment consists in the computation of 817,101 ray paths (the full set of seismic events of year 1999) on 16 processors. All machines run Globus [9] and we use MPICH-G2 [8] as message passing library. Table 1 shows the resources used in the experiment. They are located at two geographically distant sites. Processors 1 to 6 (standard PCs with Intel PIII and AMD Athlon XP), and 7, 8 (two Mips processors of an SGI Origin 2000) are in the same premises, whereas processors 9 to 16 are taken from an SGI Origin 3800 (Mips processors) named *leda*, at the other end of France. The input data set is located on the PC named *dinadan* at the first site.

Machine	CPUs	Type	μ	Rating
dinadan	1	PIII/933	0.009288	1
pellinore	2	PIII/800	0.009365	0.99
caseb	3	XP1800	0.004629	2
sekhmet	4	XP1800	0.004885	1.90
merlin	5, 6	XP2000	0.003976	2.33
seven	7, 8	R12K/300	0.016156	0.57
leda	9–16	R14K/500	0.009677	0.95

Table 1. Processors used as computational nodes in the experiment.

Table 1 indicates the processors speeds observed from a series of benchmarks we performed on our application. The column μ indicates the number of seconds needed to compute one ray (the lower, the better). The associated rating is simply a more intuitive indication of the processor speed (the higher, the better): it is the inverse of μ normalized with respect to a rating of 1 arbitrarily chosen for the Pentium III/933. When several identical processors are present on a same computer (5, 6 and 9–16) the average performance is reported.

The network links throughputs between the root processor (*dinadan*) and the other nodes are reported in Table 2 assuming a linear communication cost. The column λ indicates the time in seconds needed to receive one data element from the root processor.

Machine	λ
dinadan	0
caseb	$1.00 \cdot 10^{-5}$
pellinore	$1.12 \cdot 10^{-5}$
sekhmet	$1.70 \cdot 10^{-5}$
seven	$2.10 \cdot 10^{-5}$
leda	$3.53 \cdot 10^{-5}$
merlin	$8.15 \cdot 10^{-5}$

Table 2. Measured network bandwidths (λ is in s/ray) sorted in descending order.

Notice that *merlin,* with processors 5 and 6, though geographically close to the root processor, has the smallest bandwidth because it was connected to a 10 Mbit/s hub during the experiment whereas all others are connected to fast-ethernet switches.

5.2. Results

The experimental results of this section evaluate two aspects of the study. The first experiment compares an unbalanced execution (that is the original program without any source code modification) to what we predict to be the best balanced execution. The second experiment evaluates the execution performances with respect to the two processors ordering policies, that is bandwidths in descending or ascending order.

Original application

Figure 2 reports performance results obtained with the original program, in which each processor receives an equal amount of data. We had to choose an ordering of the processors, and from the conclusion given in Section 4.4, we ordered processors by descending bandwidth.

Not surprisingly, the processors end times largely differ, exhibiting a huge imbalance, with the earliest processor finishing after 259 s and the latest after 853 s.

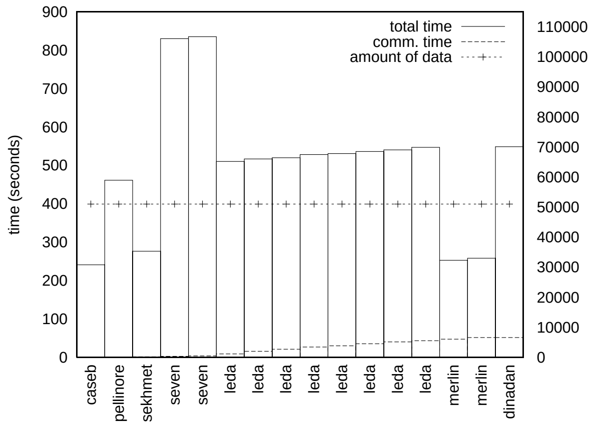

Figure 2. Original program execution (uniform data distribution).

Load-balanced application

In the second experiment we evaluate our load-balancing strategy. We made the assumption that the computation and communication cost functions were affine and increasing. This assumption allowed us to use our guaranteed heuristic. Then, we simply replaced the `MPI_Scatter` call by a `MPI_Scatterv` parameterized with the distribution computed by the heuristic. With such a large number of rays, Algorithm 1 takes 15 minutes to run on a Celeron 1.2 GHz whereas the heuristic execution, using `pipMP` [7, 17], is instantaneous and has an error relative to the optimal solution of less than $6 \cdot 10^{-6}$!

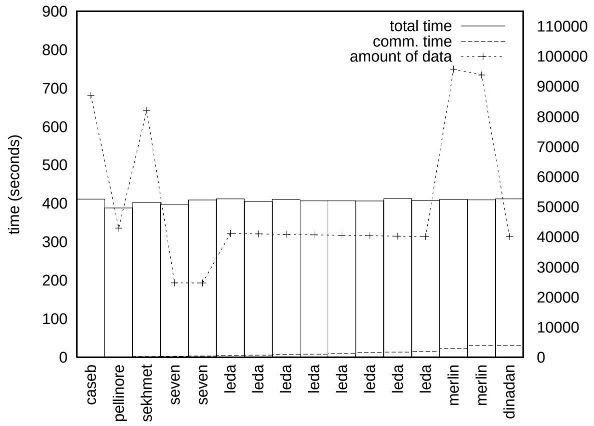

Figure 3. Load-balanced execution with nodes sorted by descending bandwidth.

Results of this experiment are presented on Figure 3. The execution appears well balanced: the earliest and latest finish times are 405 s and 430 s respectively, which represents a maximum difference in finish times of 6% of the total duration. By comparison to the performances of the original application, the gain is significant: the total execution duration is approximately half the duration of the first experiment.

Ordering policies

We now compare the effects of the ordering policy. Results presented on Figure 3 were obtained with the descending bandwidth order. The same execution with processors sorted in ascending bandwidth order is presented on Figure 4.

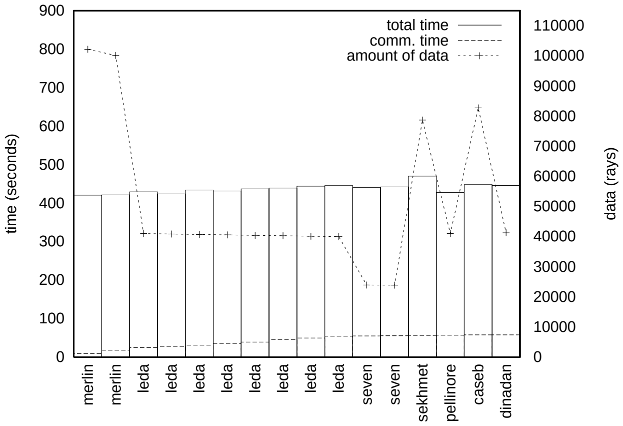

Figure 4. Load-balanced execution with nodes sorted by ascending bandwidth.

The load balance in this execution is acceptable with a maximum difference in ending times of about 10% of the total duration (the earliest and latest processors finish after 437 s and 486 s). As predicted, the total duration is longer (56 s) than with the processors in the reverse order. Though the load was slightly less balanced than in the first experiment (because of a peak load on *sekhmet* during the experiment), most of the difference comes from the idle time spent by processors waiting before the actual communication begins. This clearly appears on Figure 4: the surface of the bottom area delimited by the dashed line (the "stair effect") is bigger than in Figure 3.

6. Related work

Many research works address the problem of load-balancing in heterogeneous environments, but most of them consider dynamic load-balancing. As a representative of the dynamic approach, the work of [11] is strongly related to our problem. In this work, a library allows the programmer to produce per process load statistics during execution, and the information may be then used to decide to redistribute arrays from one iteration to the other. However, the dynamic load evaluation and data redistribution make the execution suffer from overheads that can be avoided with a static approach.

The static approach is used in various contexts. It ranges from data partitioning for parallel video processing [1] to finding the optimal number of processors in linear algebra algorithms [3].

Some works are closer to ours. The distribution of loops for heterogeneous processors so as to balance the work-load is studied in [6] and, in particular, the case of independent iterations, which is equivalent to a scatter operation. However, computation and communication cost functions are affine. A load-balancing solution is first presented for heterogeneous processors, only when no network contentions occur. Then, the contention is taken into account but for homogeneous processors only. In the framework of the Apples project, [5] discusses the load-balance of an iterative solver making stencil computations. They suggest linear programming techniques to compute a distribution, but they actually use a less precise though simplest solution by solving linear equations.

Another way to load-balance a scatter operation is to implement it following the master/slave paradigm. The general framework studied in [2] for static load-balancing could serve this purpose, but the code rewriting in this case becomes far more complex.

7. Conclusion

In this paper we partially addressed the problem of adapting to the Grid existing parallel applications designed for parallel computers. We studied the static load-balancing of scatter operations when no assumptions are made on the processor speeds or on the network links bandwidth. We presented two solutions to compute load-balanced distributions: a general and exact algorithm, and a heuristic far more efficient for simple cases (affine computation and communication times). We also proposed a policy on the processor ordering: we order them in decreasing order of the network bandwidth they have with the root processor. On our target application, our experiments showed that replacing `MPI_Scatter` by `MPI_Scatterv` calls used with clever distributions leads to great performance improvement at low cost.

Acknowledgments

A part of the computational resources used are taken from the Origin 3800 of the CINES (http://www.cines.fr/).

We want to thank them for letting us have access to their machines.

References

[1] D. T. Altilar and Y. Parker. Optimal scheduling algorithms for communication constrained parallel processing. In *Euro-Par 2002 Parallel Processing*, volume 2400 of *LNCS*, pages 197–206. Springer-Verlag, Aug. 2002.

[2] C. Banino, O. Beaumont, A. Legrand, and Y. Robert. Scheduling strategies for master-slave tasking on heterogeneous processor Grids. In *Applied Parallel Computing: Advanced Scientific Computing: 6th International Conference (PARA'02)*, volume 2367 of *LNCS*, pages 423–432. Springer-Verlag, June 2002.

[3] J. G. Barbosa, J. Tavares, and A. J. Padilha. Linear algebra algorithms in heterogeneous cluster of personnal computers. In *9th Heterogeneous Computing Workshop (HCW'00)*, pages 147–159. IEEE Computer Society, May 2000.

[4] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Bandwidth-centric allocation of independent tasks on heterogeneous platforms. In *International Parallel and Distributed Processing Symposium (IPDPS'02)*. IEEE Computer Society, Apr. 2002.

[5] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (SC'96)*. ACM Press, Nov. 1996.

[6] M. Cierniak, M. J. Zaki, and W. Li. Compile-time scheduling algorithms for heterogeneous network of workstations. *The Computer Journal, special issue on Automatic Loop Parallelization*, 40(6):356–372, Dec. 1997.

[7] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, 1988.

[8] I. Foster and N. T. Karonis. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In SC 1998 [18].

[9] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[10] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Aug. 1998.

[11] W. George. Dynamic load-balancing for data-parallel MPI programs. In *Message Passing Interface Developer's and User's Conference (MPIDC'99)*, Mar. 1999.

[12] M. Grunberg, S. Genaud, and C. Mongenet. Parallel seismic ray-tracing in a global earth mesh. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02)*, volume 3, pages 1151–1157. CSREA Press, June 2002.

[13] P. Husbands and J. C. Hoe. MPI-StarT: Delivering network performance to numerical applications. In SC 1998 [18].

[14] N. T. Karonis, B. R. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *International Parallel and Distributed Processing Symposium (IPDPS'00)*, pages 377–384. IEEE Computer Society, May 2000.

[15] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. *ACM SIGPLAN Notices*, 34(8):131–140, 1999.

[16] MPI Forum. MPI: A message passing interface standard. Technical report, University of Tennessee, Knoxville, TN, USA, June 1995.

[17] PIP/PipLib. `http://www.prism.uvsq.fr/~cedb/bastools/piplib.html`.

[18] *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (SC'98)*. IEEE Computer Society, Nov. 1998.

[19] R. Wolski, N. T. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computing Systems*, 15(5-6):757–768, Oct. 1999.

Biographies

Stéphane Genaud received a PhD in computer science from Louis Pasteur University of Strasbourg (France) in 1997. He has been an associate professor at Robert Schumann University since 1998, and a researcher in the ICPS-LSIIT laboratory in Strasbourg. His research interests involve languages and methods for parallel programming, cluster and Grid computing. Since fall 2001, he has been heading the TAG project (http://grid.u-strasbg.fr/) which study the performance of parallel scientific applications running on Grids.

Arnaud Giersch is currently a PhD student in the ICPS-LSIIT laboratory at Louis Pasteur University of Strasbourg (France). He is mainly interested in methods for parallel programming computational grids.

Frédéric Vivien obtained a PhD in computer science from École normale supérieure de Lyon (France) in 1997. From 1998 until 2002, he had been an associate professor at Louis Pasteur University of Strasbourg (France), and a researcher in the ICPS-LSIIT laboratory. He spent the year 2000 working in the Computer Architecture Group of the MIT Laboratory for Computer Science (Cambridge, USA). He is currently a full researcher from INRIA, working in the Computer Science Laboratory LIP at École normale supérieure de Lyon (France). His main research interests are scheduling techniques, parallel algorithms for clusters and grids, and automatic compilation/parallelization techniques.