Laboratoire de l'Informatique du Parallélisme

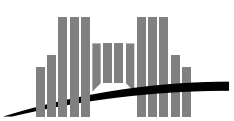
École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 5668

Load-Balancing Scatter Operations for Grid Computing

Stéphane Genaud,
Arnaud Giersch,
Frédéric Vivien

March 2003

Research Report N° 2003-17



École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

Load-Balancing Scatter Operations for Grid Computing

Stéphane Genaud, Arnaud Giersch, Frédéric Vivien

March 2003

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 study the replacement of scatter operations with parameterized scatters, allowing custom distributions 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 illustrate the benefits obtained from our load-balancing.

Keywords: parallel programming, grid computing, heterogeneous computing, load-balancing, scatter operation.

Résumé

Nous présentons des solutions pour équilibrer statiquement la charge des opérations « scatter » dans les codes parallèles exécutés sur les grilles. Notre stratégie d'équilibrage de charge est basée sur la modification des distributions des données utilisées dans les opérations scatter. Nous étudions la substitution des opérations scatter par des scatters paramétrés, permettant des distributions de données adaptées. L'article présente : 1) un algorithme général qui trouve une distribution optimale des données entre les processeurs; 2) une heuristique garantie plus rapide s'appuyant sur des hypothèses sur les communications et les calculs; 3) une politique d'ordonnancement des processeurs. Des résultats expérimentaux avec un code scientifique MPI illustre les gains obtenus par notre équilibrage de charge.

Mots-clés: programmation parallèle, grille de calcul, calcul hétérogène, équilibrage de charge, opération scatter.

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* [11] 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.

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 [21] 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 [19], MPI-StarT [17], and MPICH-G2 [9], 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 [18]. Making the communication library aware of the precise network topology is not easy: MPICH-G2 queries the underlying Globus [10] 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 really 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, the scatter operation may be used 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 [14]) assumes an homogeneous set of processors (the implicit target being a parallel computer). There is one MPI process per processor. 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 mod P) items distribution is not shown here. Figure 1 shows a potential execution of this communication operation, with P_4 as root processor.

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

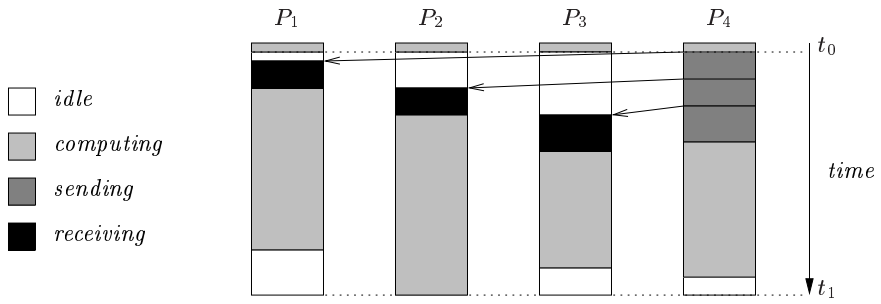


Figure 1: A scatter communication followed by a computation phase.

network cards. As the root processor sends data to processors in turn[1] a receiving processor actually begins its communication after all previous 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

In this section, we present different ways to solve the optimal data distribution problem. After briefly presenting our framework, we give two dynamic programming algorithms, the second one being more efficient than the first one, but under some additional hypotheses on the cost functions. We finish by presenting a guaranteed heuristic using linear programming that can be used to quickly find a very good approximation when the cost functions are affine.

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 [25]) running aside the application could be queried just before a scatter operation to retrieve the instantaneous grid characteristics.

3.1 Framework

In this paragraph, we introduce some notations, as well as the cost model used to further derive the optimal data distribution.

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

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

share of the data. Thus, P_i ends its processing at time:

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

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

$$T = \max_{1 \leq i \leq p} T_i = \max_{1 \leq i \leq p} \left(\sum_{j=1}^{i} T_{\text{comm}}(j, n_j) + T_{\text{comp}}(i, n_i) \right), \tag{2}$$

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

3.2 An exact solution by dynamic programming

In this section we present two dynamic programming algorithms to compute the optimal data distribution. The first one only assumes that the cost functions are non-negative. The second one presents some optimizations that makes it perform far better, but under the further hypothesis that the cost functions are increasing.

Basic algorithm

We now study Equation (2). The overall execution time is the maximum of the execution time of P_1, and of the other processors:

$$T = \max \Bigg(T_{\text{comm}}(1, n_1) + T_{\text{comp}}(1, n_1), \max_{2 \leq i \leq p} \left(\sum_{j=1}^{i} T_{\text{comm}}(j, n_j) + T_{\text{comp}}(i, n_i) \right) \Bigg).$$

Then, one can remark that all the terms in this equation contain the time needed for the root processor to send P_1 its data. Therefore, Equation (2) can be written:

$$T = T_{\text{comm}}(1, n_1) + \max \left(T_{\text{comp}}(1, n_1), \max_{2 \leq i \leq p} \left(\sum_{j=2}^{i} T_{\text{comm}}(j, n_j) + T_{\text{comp}}(i, n_i) \right) \right).$$

So, we notice that the time to process n data on processors 1 to p is equal to the time taken by the root to send n_1 data to P_1 plus the maximum of 1) the time taken by P_1 to process its n_1 data; 2) the time for processors 2 to p to process $n - n_1$ data. This leads to the dynamic programming Algorithm 1 presented on page 5 (the distribution is expressed as a list, hence the use of the list constructor cons). In Algorithm 1, cost$[d, i]$ denotes the cost of the processing of d data items over the processors P_i through P_p. solution$[d, i]$ is a list describing a distribution of d data items over the processors P_i through P_p which achieves the minimal execution time cost$[d, i]$.

Algorithm 1 has a complexity of $O(p \cdot n^2)$, which may be prohibitive. But Algorithm 1 only assumes that the functions $T_{\text{comm}}(i, x)$ and $T_{\text{comp}}(i, x)$ are non-negative and null whenever $x = 0$.

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

function compute-distribution(n, p)

```
1:  solution[0, p] ← cons(0, NIL)
2:  cost[0, p] ← 0
3:  for d ← 1 to n do
4:     solution[d, p] ← cons(d, NIL)
5:     cost[d, p] ← T_comm(p, d) + T_comp(p, d)
6:  end for
7:  for i ← p − 1 downto 1 do
8:     solution[0, i] ← cons(0, solution[0, i + 1])
9:     cost[0, i] ← 0
10:    for d ← 1 to n do
11:       (sol, min) ← (0, cost[d, i + 1])
12:       for e ← 1 to d do
13:          m ← T_comm(i, e) + max(T_comp(i, e), cost[d − e, i + 1])
14:          if m < min then
15:             (sol, min) ← (e, m)
16:          end if
17:          solution[d, i] ← cons(sol, solution[d − sol, i + 1])
18:          cost[d, i] ← min
19:       end for
20:    end for
21: end for
22: return (solution[n, 1], cost[n, 1])
```

Optimized algorithm

If we now make the assumption that $T_{\text{comm}}(i, x)$ and $T_{\text{comp}}(i, x)$ are increasing with x, we can make some optimizations on the algorithm. These optimizations consist in reducing the bounds of the inner loop (e-loop, lines 12–19 of Algorithm 1). Algorithm 2, on page 6, presents these optimizations.

Let us explain what changed between the two algorithms. For the following, remember the hypothesis that $T_{\text{comm}}(i, x)$ and $T_{\text{comp}}(i, x)$ are increasing with x. As $T_{\text{comm}}(i, x)$ and $T_{\text{comp}}(i, x)$ are non-negative, $\text{cost}[x, i]$ is obviously increasing too, and thus $\text{cost}[d - x, i]$ is decreasing with x. The purpose of the e-loop is to find sol in $[0, d]$ such that $T_{\text{comm}}(i, sol) + \max(T_{\text{comp}}(i, sol), \text{cost}[d - sol], i + 1)$ is minimal. We try 1) to reduce the upper bound of this loop, and 2) to increase the lower bound.

Let e_{max} be the smallest integer such that $T_{\text{comp}}(i, e_{max}) \geq \text{cost}[d - e_{max}, i + 1]$. For all $e \geq e_{max}$, we have $T_{\text{comp}}(i, e) \geq T_{\text{comp}}(i, e_{max}) \geq \text{cost}[d - e_{max}, i + 1] \geq \text{cost}[d - e, i + 1]$, so $\min_{e \geq e_{max}}(T_{\text{comm}}(i, e) + \max(T_{\text{comp}}(i, e), \text{cost}[d - e, i + 1]))$ equals to $\min_{e \geq e_{max}}(T_{\text{comm}}(i, e) + T_{\text{comp}}(i, e))$. As $T_{\text{comm}}(i, e)$ and $T_{\text{comp}}(i, e)$ are both increasing with e, $\min_{e \geq e_{max}}(T_{\text{comm}}(i, e) + T_{\text{comp}}(i, e))$ equals to $T_{\text{comm}}(i, e_{max}) + T_{\text{comp}}(i, e_{max})$. By using a binary search to find e_{max} (lines 16–26 of Algorithm 2), and by taking care for the cases when e_{max} falls before 0 (line 12) or after d (line 14), we can reduce the upper bound of the e-loop. To take advantage of this information, the direction of the loop must also be inverted. Besides that, we know that inside the loop, $\text{cost}[d - e, i + 1]$ is always greater than $T_{\text{comp}}(i, e)$, so the max in the computation of m can be

Algorithm 2 Compute an optimal distribution of n data over p processors (optimized version)

function compute-distribution(n, p)

```
1: solution[0, p] ← cons(0, NIL)
2: cost[0, p] ← 0
3: for d ← 1 to n do
4:    solution[d, p] ← cons(d, NIL)
5:    cost[d, p] ← T_comm(p, d) + T_comp(p, d)
6: end for
7: for i ← p − 1 downto 1 do
8:    solution[0, i] ← cons(0, solution[0, i + 1])
9:    cost[0, i] ← 0
10:   for d ← 1 to n do
11:      if T_comp(i, 0) ≥ cost[d, i + 1] then
12:         (sol, min) ← (0, T_comm(i, 0) + T_comp(i, 0))
13:      else if T_comp(i, d) < cost[0, i + 1] then
14:         (sol, min) ← (d, T_comm(i, d) + cost[0, i + 1]))
15:      else
16:         (e_min, e_max) ← (0, d)
17:         e ← ⌊d/2⌋
18:         while e ≠ e_min do
19:            if T_comp(i, e) < cost[d − e, i + 1] then
20:               e_min ← e
21:            else
22:               e_max ← e
23:            end if
24:            e ← ⌊(e_min + e_max)/2⌋
25:         end while
26:         (sol, min) ← (e_max, T_comm(i, e_max) + T_comp(i, e_max))
27:      end if
28:      for e ← sol − 1 downto 0 do
29:         m ← T_comm(i, e) + cost[d − e, i + 1]
30:         if m < min then
31:            (sol, min) ← (e, m)
32:         else if cost[d − e, i + 1] ≥ min then
33:            break
34:         end if
35:      end for
36:      solution[d, i] ← cons(sol, solution[d − sol, i + 1])
37:      cost[d, i] ← min
38:   end for
39: end for
40: return (solution[n, 1], cost[n, 1])
```

avoided (line 29).

We cannot proceed the same way to increase the lower bound of the e-loop. We can however remark that, as the loop has been inverted, e is decreasing, so $\text{cost}[d-e, i+1]$ is increasing. If $\text{cost}[d-e, i+1]$ becomes greater than or equal to min, then for all $e' < e$, we have $\text{cost}[d-e', i+1] \geq \text{cost}[d-e, i+1] \geq min$, and as $T_{\text{comm}}(i, x)$ is non-negative, $T_{\text{comm}}(i, e') + \text{cost}[d-e', i+1] \geq min$. The iteration can thus be stopped, hence the **break** (line 33).

In the worst case, the complexity of Algorithm 2 is the same than for Algorithm 1, i.e. $O(p \cdot n^2)$. In the best case, it is $O(p \cdot n)$. We implemented both algorithms, and in practice Algorithm 2 is far more efficient.

In spite of these optimizations, running the implementation of Algorithm 2 is still time-consuming. That is why we now present a more efficient heuristic valid for simple cases.

3.3 A guaranteed heuristic using linear programming

In this section, we consider the realistic but less general case when all communication and communication times are affine functions. This new assumption enables us to code our problem as a linear program. Furthermore, from the linear programming formulation we derive an efficient and guaranteed heuristic.

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

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

We must solve this linear program in integer because we need an integer solution. The integer resolution is however very time-consuming.

Fortunately, a nice workaround exists which provides a close approximation: we can solve the system 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:

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

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:

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

which implies Equation (4) knowing that $T_{opt} \leq T'$, $T \leq T_{opt}$, and finally that:

$$T = \max_{1 \leq i \leq p} (\sum_{j=1}^{i} T_{\text{comm}}(j, n_j) + T_{\text{comp}}(i, n_i)).$$

Rounding scheme

Our rounding scheme is trivial: first we round, to the nearest integer, the non integer 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 processor

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 one of the previous algorithms and heuristic. 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. This case study will enable us to define a policy on the order in which the processors must receive their data.

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, we assume that 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.

We show in Section 4.3 that, in this simple case, the processor ordering leading to the shortest execution time is quite simple. 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 a guaranteed heuristic 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 duration 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 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 the execution duration, t, and the number of data processor P_i must process, 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, then $T_i = T_{i-1} = t$, $n_i = \frac{\mu_{i-1}}{\lambda_i+\mu_i} \cdot n_{i-1}$, 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 $T_{\text{comp}}(i, n)$ which 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 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 satisfying the condition $\lambda_i > D(P_{i+1}, \ldots, P_p)$ 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 processor 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 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 processor 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, and thus:

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

On the opposite, if there exists at least one 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:

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

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} is 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 $T_{\text{comm}}(i,n)$ or $T_{\text{comp}}(i,n)$).

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 orderings 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:

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

(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:

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

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. Therefore, 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 [10] and we use MPICH-G2 [9] 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.

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

Machine	CPU #	Type	μ (s/ray)	Rating	λ (s/ray)
dinadan	1	PIII/933	0.009288	1	0
pellinore	2	PIII/800	0.009365	0.99	$1.12 \cdot 10^{-5}$
caseb	3	XP1800	0.004629	2	$1.00 \cdot 10^{-5}$
sekhmet	4	XP1800	0.004885	1.90	$1.70 \cdot 10^{-5}$
merlin	5, 6	XP2000	0.003976	2.33	$8.15 \cdot 10^{-5}$
seven	7, 8	R12K/300	0.016156	0.57	$2.10 \cdot 10^{-5}$
leda	9–16	R14K/500	0.009677	0.95	$3.53 \cdot 10^{-5}$

Table 1 indicates the processors speeds as well as the network links throughputs between the root processor (*dinadan*) and the other nodes. The values come 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 and the other nodes are reported in column λ assuming a linear communication cost. It indicates the time in seconds needed to receive one data element from the root processor. Considering linear communication costs is sufficiently accurate in our case since the network latency is negligible compared to the sending time of the data blocks.

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 were 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 our processor ordering policy (the processors are ordered in descending order of their bandwidths) by comparing this policy to the opposite one (the processors are ordered in ascending order of their bandwidths).

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.

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 more than two days of work (we interrupted it before its completion) and Algorithm 2 takes 6 minutes to run on a Pentium III/933 whereas the heuristic execution, using `pipMP` [8, 22], is instantaneous and has an error relative to the optimal solution of less than $6 \cdot 10^{-6}$!

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.

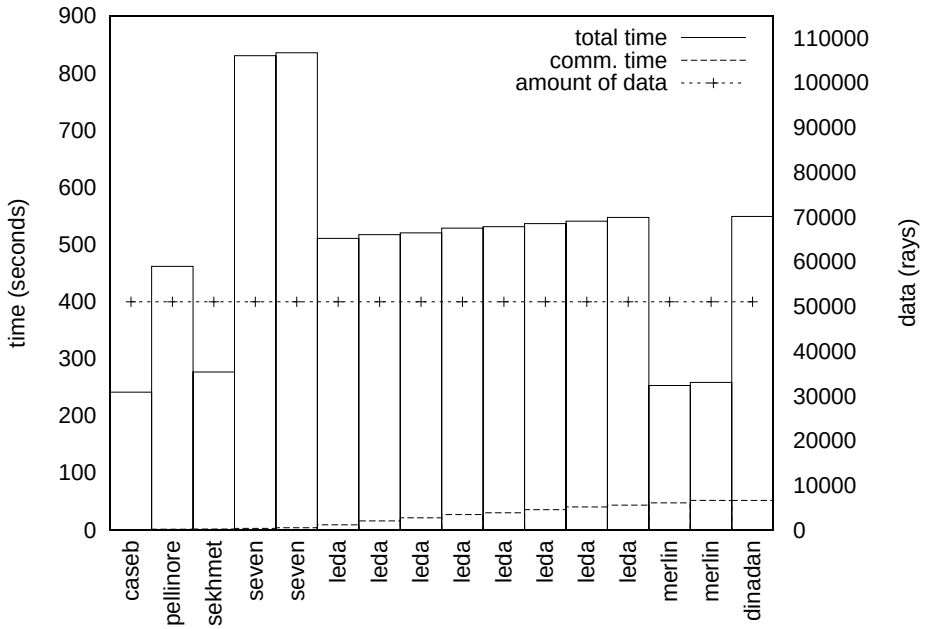


Figure 2: Original program execution (uniform data distribution).

Ordering policy

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.

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 [12] 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]. More generally, many results have been produced by the divisible load theory for various network topologies (see [6] for an overview). A part of our framework fits in the divisible load theory since we

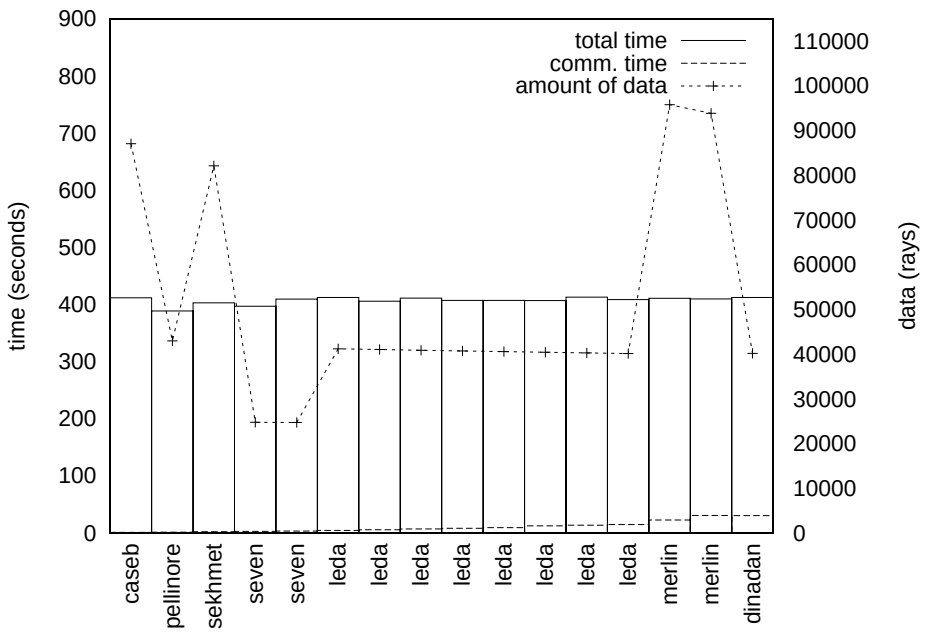


Figure 3: Load-balanced execution with nodes sorted by descending bandwidth.

consider a data set (the load) whose data items are independent. Our hardware model corresponds to the single-level tree network studied in [20]. Despite similarities for some results (for example the best processor ordering policy is also to order processors by decreasing bandwidth in [20]) this framework has major differences with ours. First, the divisible load theory considers rational load shares, with the important consequence that the optimal completion time is the same for all processors. Secondly, the communication and computation costs are linear in the load. Thirdly, when heterogeneity is considered, differences between processors or network links are also expressed as a ratio to a standard processor or link (closed form solutions for single-level tree networks are established in the homogeneous case only).

Some works are closer to ours. The distribution of loops for heterogeneous processors so as to balance the work-load is studied in [7] 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.

Another way to load-balance a scatter operation is to implement it following the master/slave paradigm. This can be done using a dynamic approach as in [16] where the MW library [13] is used to implement their solution. For a static load-balancing, the general framework studied in [2] could serve this purpose. More specifically and close to our framework, a polynomial-time algorithm is also presented in [5] for allocating independent tasks on an heterogeneous single-level tree network. The main difference with our work is that they allow communication/computation overlapping. In our work, we chose to keep the same communication structure as the original program, in order to have feasible automatic code transformation rules. Hence we do not consider interlacing computation and communication phases. At a higher level, the work from [24] considers finding the best locations for the master and the slaves given their computation capabilities and the network topology using a network flow approach. Nonetheless, using the master/slave paradigm induces a far more complex

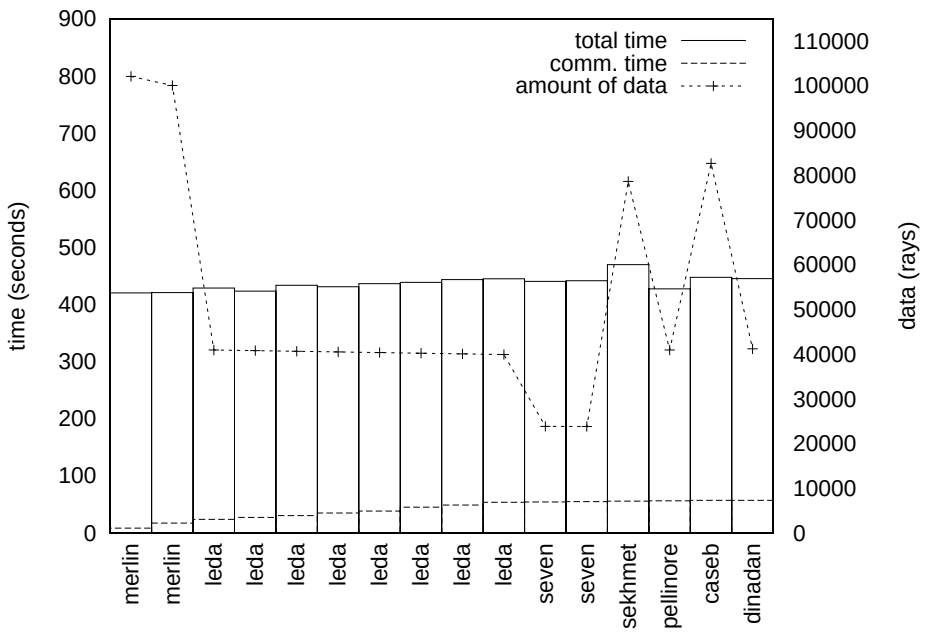


Figure 4: Load-balanced execution with nodes sorted by ascending bandwidth.

code rewriting process.

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 personal computers. In HCW 2000 [15], pages 147–159.

[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)*, page 0067. IEEE Computer Society Press, Apr. 2002.

[5] O. Beaumont, A. Legrand, and Y. Robert. A polynomial-time algorithm for allocating independent tasks on heterogeneous fork-graphs. In *17th International Symposium on Computer and Information Sciences (ISCIS XVII)*, pages 115–119. CRC Press, Oct. 2002.

[6] V. Bharadwaj, D. Ghose, and T. G. Robertazzi. Divisible load theory: A new paradigm for load scheduling in distributed systems. *Cluster Computing*, 6(1):7–17, Jan. 2003.

[7] 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.

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

[9] I. Foster and N. T. Karonis. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In SC 1998 [23].

[10] 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.

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

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

[13] J.-P. Goux, S. Kulkarni, J. Linderoth, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *9th IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, pages 43–50. IEEE Computer Society Press, Aug. 2000.

[14] 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.

[15] *9th Heterogeneous Computing Workshop (HCW'00)*. IEEE Computer Society Press, May 2000.

[16] E. Heymann, M. A. Senar, E. Luque, and M. Livny. Self-adjusting scheduling of master-worker applications on distributed clusters. In *Euro-Par 2001 Parallel Processing*, volume 2150 of *LNCS*, pages 742–751. Springer-Verlag, Aug. 2001.

[17] P. Husbands and J. C. Hoe. MPI-StarT: Delivering network performance to numerical applications. In SC 1998 [23].

[18] 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 Press, May 2000.

[19] 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, Aug. 1999.

[20] H. J. Kim, G.-I. Jee, and J. G. Lee. Optimal load distribution for tree network processors. *IEEE Transactions on Aerospace and Electronic Systems*, 32(2):607–612, Apr. 1996.

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

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

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

[24] G. Shao, F. Berman, and R. Wolski. Master/slave computing on the Grid. In HCW 2000 [15], pages 3–16.

[25] 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.