A Fast Distributed Mapping Algorithm

Jacques E. Boillat Peter G. Kropf
Institute of Informatics and Applied Mathematics
University of Berne
Länggassstrasse 51, CH–3012 BERNE

Abstract

Generating an efficient program for a parallel computer requires that the distribution of the processes on the processors comprising the parallel computer is most optimal. This paper presents a new method for a load balanced and communication optimized process distribution onto an arbitrary processor (network) topology. As opposed to many other approaches for this problem, the presented algorithm is fully distributed and based on a purely local method. It has shown to be much faster compared to the classical methods like simulated annealing, heuristic search, etc.

1 The Mapping Problem

The so called *mapping problem* is fundamental in distributed or parallel computing. From a topological viewpoint, the mapping may be defined as follows:

> Is there a mapping of a system of communicating processes onto a processor network such that the neighboring processes are assigned to neighboring processors ?

Since mapping a set of communicating processes onto processor networks is known to be NP–complete [Bok81], it makes no sense to try and develop exact algorithms for solving the problem. Furthermore, as such mapping algorithms should be integrated into distributed operating systems, they should be able to produce suboptimal solutions very quickly. The mapping algorithm presented here (called the **Mapper**), is fully distributed and delivers optimal or suboptimal solutions in a short time. It has been implemented for mapping a static system of communicating processes onto a processor network with distributed memory architecture. The Mapper has been implemented in **occam**, and can be run on an arbitrary Transputer network.

A parallel program expressed as a set of communicating sequential processes [Hoa85] can be represented as a graph, where each process forms a vertex and each communication path between any two processes forms an edge. Similarly a processor network can be defined

as a graph, with the processors being vertices and the interconnections or communication links being edges. The mapping problem is thus the problem of embedding the software graph into the hardware graph. Because the mapping should deliver a distribution of the processes such that the most efficient execution results, there are two optimization demands to be met :

- **Load balancing:** The processes have to be distributed such that the overall system load is well balanced, i.e. that the load caused by the processes is distributed evenly over all the processors.

- **Communication minimization:** The communication capacity between any two processors should be used optimally, i.e. the overall communication should be distributed as evenly as possible over all communication links.

The demand for communication optimization means that neighboring processes should be mapped onto neighboring processors. In the case of a communication between two processes which are not placed on neighboring processors, and for example more than one communication link has to be used, higher communication and processor costs are incurred. Furthermore on transient processors, such a communication also causes load, because it has to be routed on to its destination. If it is not possible to map neighboring processes onto neighboring processors, the processes should be placed such that the overall communication costs are minimal.

Mapping Algorithms

There are many different approaches to solve the mapping problem. Among these are simulated annealing [DSS88, FKW87], cardinality based algorithms [Bok81], heuristic search [Per89, BKMW87, She88] and evolutionary algorithms [MGK87, BKW88].

Common to all these methods is their global approach: the optimization demands are controlled by central decision making mechanisms, e.g. the temperature decrease in the case of simulated annealing. Algorithms based on these methods can yield good results, however usually take far too much time to find a satisfactory solution if the graphs are not very small. Some such algorithms can be applied reasonably for larger numbers of processes when certain restrictions are met, e.g. the cardinality based algorithm [Bok81] for hardware and software graphs of identical sizes. To speed up the mapping using such approaches the algorithms can be parallelized, but only to an extent, because they must always maintain communication with the decision making center. This means that any implementation of such algorithms, be it sequential or parallel, is always a centralistic one.

Distributed Mapping

The Mapper may be run in parallel on the target hardware itself. It is based on the following principles:

- fully distributed method
- local, nearest neighbor optimization leads to a global optimization
- neighboring processors exchange information about their load and communication costs
- processes are moved according to the optimization demands
- making use of the parallel hardware itself to increase the speed of the mapping

The algorithm is made up of identical communicating processes running on each processor. After a certain number of iteration steps an optimal or suboptimal solution to the mapping problem is found. At each iteration step, the neighboring processors exchange the current knowledge about their load and the probable position of all the processes. Processes causing high local communication costs are chosen for being sent to neighboring processors reducing the local costs. Since the probable location of the neighboring processes of the process to be moved is known, the process can be sent to the correct processor. Fig. 1 shows the general structure of the mapping process running in parallel on each processor.

```
SEQ i = 0 FOR iterations
  SEQ
    ... exchange information and processes with direct neighbors
    ... choose process causing high local communication costs
    ... decide whether to send it to neighboring processor
    ... balance load locally : Load Balancer
```

Figure 1: General structure of the *Mapper*

2 Formal Description of the Mapping Problem

As target architectures for the definition of the mapping problem we assume the following:

- The hardware is a distributed memory architecture
- Each processor has its own private memory
- Processor interconnections are realized by communication channels or links
- The interconnection topology is fixed or statically reconfigurable

Definition 1 (Multiprocessor System) *A* multiprocessor system with distributed memory *is a pair :*

$$H = (< T, L^H >, C^H)$$

where:

- $< T, L^H >$ *is the (undirected) graph representing the network structure or topology*
  - $T = \{T_1, T_2, ..., T_n\}$ *is a set of processors*
  - L^H *is the adjacency matrix of the hardware graph*
- C^H *is the hardware communication cost matrix.*
  c^H_{ij} *denotes the costs of transfering one information unit between* T_i *and* T_j.

Definition 2 (Static System of Communicating Processes) *A* static system of communicating processes *is a triple:*

$$S = (< P, L^S >, C^S, time)$$

where:

- $< P, L^S >$ *is the (directed) graph representing the structure (topology) of a system of communicating processes*
  - $P = \{P_1, P_2, ..., P_m\}$ *is a set of processes*
  - L^S *is the adjacency matrix of the software graph*
- C^S *is the software communication cost matrix*
  $c^S_{\mu\nu}$ *denotes the number of information units passed from process* P_μ *to process* P_ν.
- $time : P \longrightarrow \mathbf{N}$
  $time(P_\mu)$ *denotes the number of atomic computational steps performed by process* P_μ.

Definition 3 (Restriction) *A* restriction *is a subset* $R \subseteq P \times T$.

$$(\mu, i) \in R$$

means that process P_μ *has to be assigned to processor* T_i.

Restrictions need to be considered, because certain processes may access peripheral resources like I/O devices attached to specific processors. Such processes must thus be placed on the correct processor.

Definition 4 (Load) *The* load *of a processor relative to a process mapping* π *is defined as*

$$load(T_i) = \sum_{\pi(P_\mu)=T_i} time(P_\mu)$$

The notion of *time* caused by a process is problematic, because it is not easy to calculate the load a process will produce. There are basically two ways of determining the load. One is to calculate the theoretical time complexity of a process, the other is to determine the absolute time a process uses by actually running it. The first one is not realistic, because the theoretical complexity cannot be computed automatically. The second is more accurate, but involves the problem of monitoring a process, i.e. the problem of measuring out the time behavior of a parallel program. Furthermore, in most cases the load of a process changes dynamically during its life. For a statical mapping algorithm it is obvious, that the load of a process should be taken as a constant, which can be defined as the mean of its expected load. To obtain realistic values for the load and the communication costs between processes, a monitoring tool like the Transputer Performance Analyser [TnT90] can be used.

Definition 5 (Optimal Mapping) *An* optimal process mapping *is a function*

$$\pi : P \longrightarrow T$$

such that:

- $\pi(P_\mu) = T_i \qquad (\forall(\mu, i) \in R)$: *The mapping takes care of the restrictions.*
- $load(T_i) \simeq \frac{\sum_{j=1}^{m} load(T_j)}{m} \qquad (\forall i = 1, ..., m)$: *The processors are* **load balanced**.
- $\sum_{\mu,\nu} c^S_{\mu.\nu} \cdot c^H_{t(P_\mu)t(P_\nu)}$ *is minimal: The communication costs are* **minimized**.

3 Mapper: the Distributed Algorithm

The distributed mapping algorithm *Mapper* performs a mapping $\pi : P \longrightarrow T$ with the following assumptions :

- The communication costs c^H_{ij} between any two processors T_i and T_j is set to

$$c^H_{ij} = \begin{cases} 1 & \text{if } L^H_{ij} = 1 \\ 2 \times D_{ij} & \text{else} \end{cases}$$

  where D_{ij} denotes the shortest distance between T_i and T_j.
- The communication costs between any two connected processes is set equally to 1.

These assumptions do not lead to any loss of generality regarding the formal definition of a mapping [1].

The *Mapper* is based on an iteration scheme consisting in three parts :

[1] The storage requirements of the processes as well as the available storage size of the processors are neglected

(1) Information exchange
(2) Communication optimization
(3) Load balancing

This scheme is run on each processor in parallel. As the optimal solution of a mapping π is usually not known (at least not in advance), and because the problem belongs to the class of NP-complete problems, the number of iterations to use is an experimental matter. However, considering load balancing only, the number of iterations are known: this will be shown in the next section. Fig. 2 shows the principle working of the *Mapper*.

```
--------------------------------------------------------
-- Mapper
--------------------------------------------------------
PROC mapper(VAL INT processor.number)
  SEQ
    ... initialization
    SEQ i = 0 FOR iterations
      SEQ
        ... exchange load information and processes
            with direct neighbors
        ... choose process with high communication costs
        ... decide whether to send it to neighbor
        ... balance load locally
            VAL init.load IS my.load :
            SEQ j = 0 FOR links
              VAL load.diff IS init.load - nb.load :
              VAL to.give IS load.diff / weight[j] :
              IF
                give > 0.0
                  SEQ
                    my.load := my.load - to.give
                    give[j] := to.give
                TRUE
                  give.to[j] := 0
        ... update
:
```

Figure 2: The distributed mapping process

At the beginning all the processes are placed on an arbitrary processor (except those associated with a restriction). At each iteration, processes are then moved to meet the optimization demands. First the most optimal local load balance is considered. The processes which are passed on to another processor are selected such that the local communication costs are reduced. Because the load balancing is treated superior to the communication

minimization, it may happen that the load is balanced and the communication demand is not fully optimal. Therefore a process causing high local communication costs is selected with a decreasing probability to be moved to another processor in order to reduce the costs.

Communication Cost Optimization

As defined previously, the communication costs between any two processes are set to $c^S_{\mu\nu} = 1$. During the mapping process there are three types of communication considered:

- **Internal communication:** A communication between two processes currently placed on the same processor. Its value is set to $c^S_{\mu\nu}$ (this is always 1).
- **Short communication:** A communication between processes currently placed on neighboring processors. Its value is set to $c^S_{\mu\nu}c^H_{ij}$ (this is always 1).
- **Long communication:** A communication between processes currently placed on distant processors, i.e. processors which are not neighbors. Its value is set to $c^S_{\mu\nu}c^H_{ij}$.

To select candidate processes to be moved to another processor an **attraction vector** is defined as follows :

- each processor knows the shortest distance to every other processor in the network.
- each processor T_i knows all communication links l^H_{ij}, T_j a neighboring processor, through which a shortest distance to all the other processors is realized.
- each processor knows the *probable* location of all processes (these locations are exchanged with the neighboring processors at each iteration).

Because only the neighbors of a processor which has moved a process to another one know the actual location of this process, the knowledge of the process locations each processor has, is not always up to date. This is because when a process P_μ has been moved, it takes some iteration steps until all the other processors also known that process P_μ has been placed elsewhere. Thus the current location information is only probable but not exact. Of course the number of iterations steps needed until a new placement induced by processor T_i becomes known to processor T_j, is proportional to the shortest distance D_{ij}.

Attraction Vector

The attraction vector defines whether and which process P_μ currently placed on processor T_i is sent to a neighboring processor. It is constructed in the following way:

Let P_μ be a process currently placed on processor T_i. Suppose that P_μ communicates with P_ν, whose probable location is T_j. Let $l^H_{ik}, k = 1...n = |T|$ be a communication link and $\{P_\nu\}_\mu = \{P_\nu | c^S_{\mu\nu} \neq 0\}$ be the set of all processes communicating with process P_μ. $F_k(\mu)$ defines the communication costs for P_μ communicating with processes in $\{P_\nu\}_\mu$

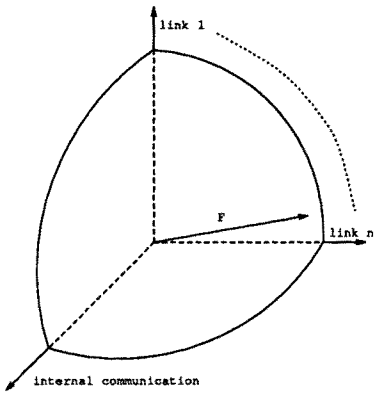

Figure 3: The attraction vector F has a high value in the direction of link n

through the communication link k. If the two processes P_μ and P_ν are placed on the same processor T_i, k is set to 0. The attraction value for a process P_μ through a communication link k is defined as follows:

$$F_0(\mu) = \begin{cases} \sum_\nu c^S_{\mu\nu} & \text{if } P_\mu \text{ and } P_\nu \text{ are placed on the same processor } T_i \\ 0 & \text{else} \end{cases}$$

$$F_k(\mu) = \begin{cases} \sum_\nu c^S_{\mu\nu} c^H_{ij} & \text{if } l^H_{ik} \text{ realizes a shortest distance } D_{ij} \\ 0 & \text{else} \end{cases}$$

Fig. 3 shows an *attraction vector* F normalized to unit. The Mapper chooses at each iteration step those processes with the highest attraction values still fulfilling the load balancing demand. If there are different processes with the same attraction value, the processes to be moved are chosen randomly. The process with the highest attraction value is sent to a neighboring processor with decreasing probability, neglecting the load balancing demand in order to assure that the algorithm does not remain in a communication suboptimum. The probability depends on the number of iterations and the current iteration number:

$$1 - \frac{\texttt{current iteration}}{\texttt{total number of iterations}}$$

4 Analysis of the Mapping Algorithm

Since the main part of the communication minimization demand is embedded in the load balancing and since the communication minimizing process moves neglecting the load balancing occur with decreasing probability, we concentrate on the Load Balancer for the analysis of the mapping algorithm.

For simplicity, we will assume that there is a very large number of processes and that

$$|P| \gg |T|$$

Furthermore, we will ignore the number of processes running on each processor and consider the load of each processor as a real number.

Let π be a mapping. A (global) *cost function* $\Gamma(\pi)$ for the load distribution may be defined as

$$\Gamma(\pi) = \sum_{i \in T} (\bar{l} - \text{load}(i))^2$$

where

$$\bar{l} = \frac{\sum_{i \in T} \text{load}(i)}{|T|}$$

In [Boi], we show that a global minimum of $\Gamma(\pi)$ can be reached by locally optimizing the function

$$\Gamma_i(\pi) = \sum_{(i,j) \in T} (\text{load}(i) - \text{load}(j))^2 \qquad \forall i \in T$$

and that an uniform distribution is reached after at most $O(|T|^3)$ iterations.

5 Results

5.1 Convergence Behavior of the distributed load balancing

The results summarized in Table 1 [Boi] show the complexity of the load balancing part of the algorithm for the complete graph F_n, the d-dimensional hypercube H_d, the path P_n, the circuit C_n, and the hypertorus $C_{n_1 \ldots n_m}$.

Graph	Complexity
F_n	$O(1)$
H_d	$O(d+1)$
P_n	$O(n^2)$
C_n	$O(n^2)$
$C_{n_1 \ldots n_m}$	$O(\max\{n_1, \ldots, n_m\}^2)$

Table 1: Convergence factor

5.2 Performance of the Mapper

A large number of tests have shown that in the case of a process system with homogeneous load, the uniform distribution is reached in more than 98% of all instances.

Table 2 shows the results of mapping a 4 × 4 mesh of processes onto a 2 × 2 dimensional processor mesh (see Figure 4).

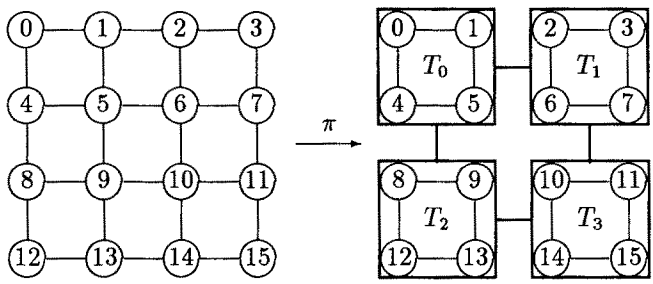

Figure 4: Optimal mapping of a 4 × 4 process mesh onto a 2 × 2 processor mesh

In the example of Table 2, all processes have the same load and all communications have the same costs. The tests have been made on a mesh of Transputers (T800, 20 MHz). For each number of iterations, 50 runs have been made. The load was optimally balanced in all runs.

number of iterations	optimal costs	mean costs	variance	success	time used
50	24	24.54	1.43	86 %	0.3 s
100	24	24.06	0.42	98 %	0.45 s
200	24	24.00	0.00	100 %	0.72 s

Table 2: Mapping a 4 × 4 process mesh onto a 2 × 2 processor mesh

Table 3 shows the results of mapping a 5 dimensional hypercube of processes onto a 3 dimensional processor hypercube. The load was balanced in all but one in 200 runs. Again for each number of iterations 50 runs have been made. The load equilibrium is reached already after about 20 steps. The other steps are needed for optimizing the communication costs.

The results show that after 800 iterations, a suboptimal placement is reached, differing in the mean only 3% from the optimal solution. 94% of the placements are optimal; they do not need any routing processes.

number of iterations	optimal costs	mean costs	variance	success	time used
200	80	99.22	22.82	54 %	1.18 s
400	80	91.50	17.99	66 %	2.31 s
600	80	85.18	14.31	88 %	3.46 s
800	80	83.02	12.19	94 %	4.62 s

Table 3: Mapping a 5-dimensional process hypercube onto a 3-dimensional processor hypercube

Note that all process placements in these two examples have been computed in a very short time. Tests done with a Petri net simulation program have shown that the Mapper is also well suited for mapping non homogeneous process systems (e.g. Petri nets) onto a multiprocessor system.

6 Conclusions

Concluding, the proposed distributed Mapper is a good alternative to the traditional mapping techniques. Indeed, the Mapper

- is very efficient and produces good suboptimal solutions
- is very simple and easy to implement
- is distributed over the whole processor network
- does not need any global synchronization: The synchronizations are restricted to local interactions between neighboring processors

We intend to apply the algorithm to reconfigurable architectures as well. The Mapper seems to be particularly well suited for reconfigurable networks, because it can adapt itself to every new switching of the processor interconnections.

Furthermore, together with a communication kernel, the Mapper is a good candidate to be included in a truely distributed operation system or in a configuring tool for parallel programs.

References

[BKMW87] J.E. Boillat, P.G. Kropf, D.Chr. Meier, and A. Wespi. An analysis and reconfiguration tool for mapping parallel programs onto transputer networks. In T. Muntean, editor, *OPPT*, Grenoble, 1987.

[BKW88] J.E. Boillat, P.G. Kropf, and K. Wyler. *Evolutionäre Algorithmen.* Technical Report IAM-PR-88205, University of Bern, Informatics, 1988.

[Boi] J.E. Boillat. Load balancing and poisson equation in a graph. To appear in Concurrency: Practice and Experience.

[Bok81] S.H. Bokhari. On the mapping problem. *IEEE Transactions on Computers*, C-30(3):550–557, 1981.

[DSS88] J.G. Donnet, M Starkey, and D.B. Skillicorn. Effective algorithms for partitioning distributed programs. In *Proceedings of the 7th Ann. Int. Phoenix Conf. on Computers and Communications*, IEE, 1988.

[FKW87] G. Fox, A. Kolawa, and R. Williams. The implementation of a dynamic load balancer. In M.T. Heath, editor, *Hypercube Multiprocessors*, page 114, SIAM, 1987.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, 1985.

[INM88a] INMOS. *OCCAM 2 Reference Manual.* Prentice-Hall, Englewood Cliffs, 1988.

[INM88b] INMOS. *Transputer Reference Manual.* Prentice-Hall, Englewood Cliffs, 1988.

[MGK87] H. Mühlenbein, M. Gorges-Schleuter, and O. Krämer. New solutions to the mapping problem of parallel systems: the evolution approach. *Parallel Computing*, 4:269–279, 1987.

[Per89] Perihelion. *The Helios Operating System.* Prentice-Hall, Englewood Cliffs, 1989.

[She88] H. Shen. Self-adjusting mapping: a heuristic mapping algorithm for mapping parallel programs onto transputer networks. In J. Wexler, editor, *Developing Transputer Applications*, IOS, Amsterdam, 1988.

[TnT90] *TNT - PFY Reference Manual.* TNT - Parallel Computing Support, Bern, 1990.