A NEW MAPPING HEURISTIC FOR LAST GENERATION MULTICOMPUTERS

Internal Report C94-09

April 1994

R. Perego
G. De Petris

A New Mapping Heuristic for Last Generation Multicomputers*

G. De Petris and R. Perego

CNUCE - C. N. R., Via S. Maria 36, 56126 Pisa, Italy

*This work has been supported by the "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of the Italian National Research Council.

2 The static mapping problem

The static mapping problem can be informally stated as the problem of assigning, at compilation time, the various communicating tasks that make–up the parallel program to a set of interconnected processors so as to minimize (or maximize) some parameters of interest in parallel execution (e.g. completion time, load balance, fault tolerance, etc. [9]). In this work we address the mapping of coarse–grained tasks those behavior can be predicted with a reasonable degree of approximation; moreover, we assume the absence of precedence relationship among the tasks which are modelled according to a *process–based* model [20].

More formally, we will tackle the static mapping problem by means of the following abstraction. Given:

1. a *parallel program* modelled by a *Task Interaction Graph* (TIG) $G_p = (N_p, E_p)$, a directed graph whose nodes, $n_i \in N_p$, represent tasks, and edges, $e_i \in E_p$, communication channels between each pair of interacting tasks,

2. a target *parallel architecture* represented by a *host graph* $G_a = (N_a, E_a)$, where each node, $n_i \in N_a$, models a processor, and each undirected edge, $e_i \in E_a$, corresponds to a physical bidirectional link connecting a pair of processing nodes,

3. all the admissible *mapping functions* $m_j : G_p \to G_a$, that associate each task, $n_i \in N_p$, to a processor, $n_k \in N_a$, and each channel, $e_i \in E_p$, to the ordered set of links through which any message impinging on e_i will be routed (when a general routing mechanism is supported by the target system, as happens for most modern multicomputers, only the tasks are assigned because the paths followed by the messages are forced by the routing rules),

4. an *evaluation function* F, defined over the set of all possible mappings, which measures mapping quality by estimating program completion time,

solving the mapping problem signifies finding $m\prime$, such that $F(m\prime) = \min_j F(m_j)$.

In general, a mapping algorithm must solve two main sub-problems: (a) the *topological variation* problem, arising when G_p and G_a graphs are not isomorphic, and, (b) the *cardinality variation* problem, arising because the number of nodes of G_p may be greater than the number of processors of G_a [6]. Due to the NP-Hardness of the problem, to date many exact and approximated solutions dealing with static process-based models of the mapping problem have been proposed [20]. Most proposals can be grouped on the basis of algorithm structure into the following four classes:

- *Branch & Bound algorithms*;
- *Local Search algorithms*;
- *Genetic approaches*;
- *Greedy algorithms.*

Branch & Bound algorithms find optimal mappings but, due to their high complexity, are suitable only for small problems. Furthermore, the optimality of the solutions reached is clearly bound by the accuracy of the evaluation function used which, in turn, must be quite simple so as to not further weigh down the cost of the algorithm. To improve the efficiency of the method, various heuristics that prune the solution tree are generally employed. Nevertheless, complexity in the worst case still remains exponential. Exact approaches are followed by Sinclair [24], Ma, Lee and Tsuchiya [18], Shen and Tsai [23]; they all use evaluation functions that take into account, at most, load balancing and remote communication delays, and report results achieved on graphs with no more than 10 processors and 23 tasks.

Local Search algorithms yield locally optimal solutions and provide means of limiting the time required to find a good solution. Starting from an admissible mapping, the solution is iteratively improved until a satisfactory assignment is found. To avoid the problem of being trapped into "bad" local optima, some techniques are often adopted which allow non-deterministic jumps to solutions outside the neighbors of the current one. Bokhari [7] uses the number of task graph edges that fall onto host graph edges (*cardinality*) as the mapping evaluation function and tests a *pairwise exchange* algorithm with probabilistic jumps on problems with up to 49 tasks. Bollinger and Midkiff [8] use a *simulating annealing* algorithm [14] to minimize remote communication delays and, in a second phase, message collisions. Results on graphs with up to 512 tasks and processors have been reported.

In *Genetic* approaches, Darwin's evolutionary mechanisms are applied to Computer Science. Mapping populations are generated by the use of operators such as *replication*, *mutation* and *selection*, directly derived from the Natural Sciences. Following this strategy, Arunkumar and Chockalingam [4], minimized communication delays for some samples of not more than 80 tasks and 16 processors. A similar approach has been used by Mühlenbein et al. in Ref. [19].

Greedy algorithms guarantee polynomial bounds on complexity because a single path from the root to a leaf of the solution tree is followed: a task is repeatedly picked and assigned to a processor until the mapping is completed. Greedy mappers generally use accurate evaluation functions to drive each assignment. Moreover, the evaluation functions must be defined over *partial* mappings as well (i.e. $m : G_p \rightarrow G_a \cup \{\perp\}$, where $m(n_i) = \perp$ means that task n_i has not yet been assigned to a processor). Antonelli et al. [2, 3], propose a greedy algorithm to minimize communication delays of CSP–like programs; Baba, Iwamoto, Yoshinaga [5] do not take into account the internal structure of the task graph (i.e. computation and communication loads), but only its shape, and report results achieved with a greedy algorithm on graphs with up to 530 nodes.

Other relevant proposals cannot be neatly classified according to the above taxonomy. Lee and Aggarwal [15], for example, propose an interesting combination of a greedy with a local search algorithm; Sadayappan, Ercal and Ramanujam [22] report on two mapping algorithms, each composed of two different phases. The first phase solves the *cardinality variation* problem by grouping the tasks into clusters, the second one assigns clusters to processors; Lo [16], continues along the lines traced by Stone [25] in transforming task and host graphs into a single network to which a *Min–Cut Max–Flow* algorithm is applied.

3 Preliminaries

3.1 Target parallel architectures

The greedy mapping algorithm discussed in this paper is designed for multicomputers composed of large ensembles of processing nodes interconnected by a k-ary n-cube direct communication network. By simply specifying the degree and dimension of the interconnection topology, we can target the mapper to deal with most existing multicomputers, ranging from 2D meshes (k-ary 2-cube) like Symult 2010, Intel Touchstone Delta and Paragon, up to hypercubes (2-ary n-cube) like nCUBE 2, Intel iPSC/2 and iPSC/860. Further assumptions made about the target parallel architecture are the following:

- the system has $N = k^n$ identical *pn*'s interconnected by l bidirectional links. The length of the shortest path (*distance*) from *pn* i to *pn* j will be indicated by $d(i,j)$;
- each processing node supports multitasking (many-to-one mappings are allowed) and is composed of a processor, a routing unit (*router*), and a local memory whose size is bounded by some constant M;
- routers exploit the deadlock-free *e-cube* routing algorithm and the *wormhole* flow-control policy [10]. Such techniques, adopted by many commercial multicomputers such as Symult 2010, nCUBE 2, Intel Touchstone Delta and Paragon, can be efficiently implemented in hardware and allow low-latency communications even for distant *pn*'s. The deterministic sequence of links used by the *e*-cube routing algorithm to deliver a message from *pn* i to *pn* j will be indicated by *path*(i,j). With wormhole routing, in the absence of network contentions, the latency $T_{wh}(i,j,L)$ of a message L bits long, sent from *pn* i to *pn* j, is:

$$T_{wh}(i,j,L) = T_c(\frac{L}{F} + (d(i,j) - 1)) \qquad (1)$$

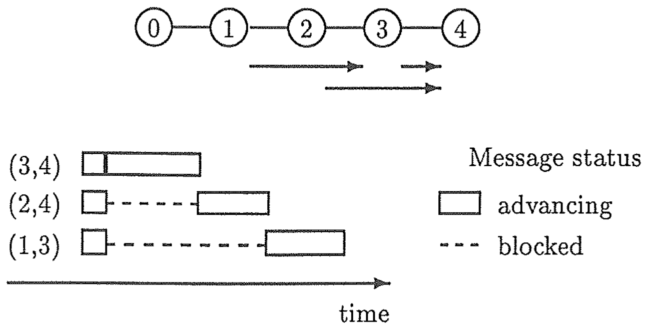

Figure 1: Network contention on a 5-ary 1-cube. The packet from 1 to 3 is blocked on node 2 until packets (2,4) and (3,4) are delivered, even if $path(1,3)$ and $path(3,4)$ are disjoint.

where T_c is the time (usually a single channel cycle) needed to transfer a *flit* (i.e. the smallest information unit into which a message is partitioned) between two adjacent *pn*'s, and F is the flit length expressed in bits [10].

- a packet of a message requiring a busy link c is blocked in the links it currently occupies until c is freed.

It is worth mentioning that the message latencies given by Eq. 1 are rarely achieved in actual programs because of packet blocking times consequent to the sharing of network resources. By comparing experimental message latencies when network traffic is heavy or unevenly distributed with respect to those modelled by Eq. 1 (Figure 1 graphically illustrates the case of a packet waiting for another one to be wholly delivered even if their paths are disjoint) one can see the importance of taking into account network contention problems in an accurate mapping scheme.

3.2 Program modelling

When a *static* approach to the mapping problem is followed, a suitable amount of information on program behavior must be initially collected. Throughout our discussion we will assume the information deduced during a *modelling step* which analyzes the parallel program or the trace records obtained during a instrumented run, and produces a deterministic description of its communication kernel in the following simple language:

PROCESS i : begins the description of task i;

INT_OP p : holds the task in a "busy state" for p CPU cycles. It models any sequence of internal operations not affecting the network. If the duration of the code section is variable, p approximates the most probable duration of the modelled code section;

SEND *dest f* : the task sends a message *f* flits long to task *dest*. Process is busy for a constant amount of time regardless of the number of flits sent or their destination, then it may execute a new instruction;

RECEIVE *sndr f* : the task receives a message *f* flits long from task *sndr*. If the message on the required channel has yet not arrived, the task executing this instruction switches into "idle state" until the arrival of the message; then the task spends a number of CPU cycles independent the number of flits or the sender;

STOP : marks the end of the task description.

Such a program description, recognized by our mapping algorithm, recalls Lo's *Temporal Communication Graph* (TCG) abstraction [17]. As Lo's TCG, it integrates the static TIG model with timing information needed to face both network contention problems and applications which exploit different communication patterns as the computation progresses. Furthermore it was chosen for the availability of a flexible simulator [21], extensively used for evaluation purposes, that measures the completion time of such programs on multicomputers having the characteristics outlined in the previous subsection.

During its execution, our mapping algorithm transforms the input TCG description according to the Synchronous Phases Model (SPM) abstraction [15]. This transformation requires the set E_p of logical communication channels to be split into subsets on the basis of temporal information on the use of each channel: a communication channel e_h is put in subset i together with channel e_j iff two messages, both occupying the network for a time longer than a predefined fraction of their whole minimal latency (modelled by Eq. 1), may be sent onto e_h and e_j. Each subset defines a different program communication phase because all the messages of phase i are assumed to be simultaneously sent and received before phase $(i+1)$ starts. The logical subdivision of the parallel program into more distinct communication phases permits reduction of the complexity of the mapping algorithm because only messages in the same phase can contend for the use of network resources.

Moreover, while devising communication phases, the algorithm deduces the following quantities needed during the mapping phase:

T: number of tasks of G_p;

a: number of channels of G_p;

w_i: average cost (in number of CPU cycles) of internal operations of task i;

m_{ij}: average number of messages sent from task i to task j;

l_{ij}: average length (in flits) of messages sent from task i to task j;

q_{ij}: average time (in CPU cycles) that task i waits for messages from j; this is computed considering the program executed on a specialized architecture isomorphic to G_p;

q_i: average time (in CPU cycles) that task i waits for messages ($q_i = \sum_{j=1,\dots,T} q_{ij}$);

P: number of communication phases;

p_{ij}^h: true iff during phase h some message is sent from task i to task j.

4 The mapping evaluation function

In order to design a good mapping heuristic, we first need to formulate an objective function which can accurately measure what is to be optimized. The mapping evaluation function (F) we propose attempts to efficiently approximate the completion time of the modelled program when assigned by a mapping function m to the processing nodes of G_a. Function F is defined following a *minimax* criterion as:

$$F(m) = \max_{n=1,\ldots,N}(R(m,n)) \tag{2}$$

where $R(m,n)$ is an estimate of the time needed for processor n to execute all the tasks assigned to it by mapping m. The use of a *minimax* formulation allows to achieve both minimization of interprocessor communication and balance of processor loads [23].

We will compute $R(m,n)$ on the basis of both computation and communication costs of all the tasks mapped onto n. A first approximation of $R(m,n), n = 1,\ldots,N$, can be easily derived from the information retrived during the modelling phase, as follows:

$$R_{comp\&idle}(m,n) = \sum_{m(i)=n} (w_i + q_i) \tag{3}$$

Eq. 3 approximates the case in which programs are executed on a isomorphic parallel system. Otherwise, communication costs depend on both the distance between the processors and on contention for paths between the processors onto which the tasks have been mapped.

From Eq. 1 it follows that in absence of collisions on networks using the wormhole strategy, some overhead is associated to task j because its partners may be mapped on non-adjacent processors. These times, deriving from the solution given by mapping m onto the topological variation problem, can be expressed as:

$$R_{distance}(m,j) = \sum_{m(i)\neq m(j)} m_{ij}(d(m(i),m(j)) - 1). \tag{4}$$

Moreover, cooperating tasks sharing the same *pn* communicate faster through local memory. Assuming communication times through local memory to be negligible, the savings in the communication times of task j due to multitasking are:

$$R_{multitasking}(m,j) = \sum_{m(i)=m(j)} \min(m_{ij}l_{ij}, q_{ij}). \tag{5}$$

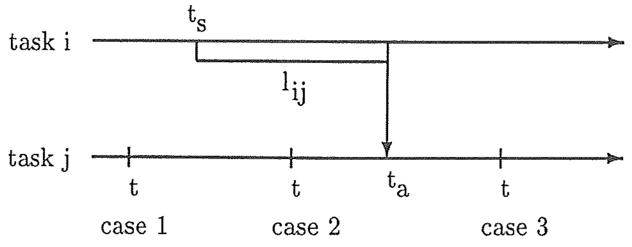

Figure 2: Message exchange between tasks mapped onto adjacent processors.

In fact, supposing for the sake of simplicity $m_{ij} = 1$, let t_s be the time in which task i sends the message to task j, t_a the time in which the message becomes available on node $m(j)$, and t the time in which task j executes the receive of the message. Considering tasks i and j placed onto different but adjacent processors ($t_a = t_s + l_{ij}$), the three cases illustrated in Figure 2 can arise:

1. $q_{ij} = (t_a - t) > l_{ij}$, i.e. task j executes the receive before t_s. If tasks i and j were assigned to the same pn, task j might be scheduled $t_a - t_s = l_{ij}$ cycles earlier;

2. $q_{ij} = (t_a - t) < l_{ij}$, i.e. task j executes the receive after t_s but before t_a. If tasks i and j were assigned to the same pn, the required message would be available as soon as task i sends it. Task j might be immediately scheduled, saving q_{ij} cycles;

3. $q_{ij} = 0$, i.e. task j executes the receive after time t_a. Task j gains nothing from sharing the processor with task i.

In all three cases Eq. 5 holds. Including overheads due to distance and savings in communication time due to multitasking, from Eq.'s 3, 4 and 5 we obtain a rather more precise estimate of completion time for node n:

$$R(m,n) = R_{comp\&idle}(m,n) + \sum_{m(j)=n} R_{distance}(m,j) - \sum_{m(j)=n} R_{multitasking}(m,j) \qquad (6)$$

Delays due to network contention are not so easy to state, even if the assumption, deriving from adoption of the Synchronous Phases Model, that all messages in the same phase start at the same moment makes the analysis more affordable.

For each communication phase h we define the relation $\equiv_h$ over the set $\{(i,j) : p_{ij}^h = 1\}$ of all the messages exchanged (all the communication channels used) during phase h as:

$$(i,j) \equiv_h (s,t) \Leftrightarrow path(m(i),m(j)) \cap path(m(s),m(t)) \neq \emptyset. \tag{7}$$

In other words $(i,j) \equiv_h (s,t)$ iff message (i,j), during phase h, can contend the use of the same link with message (s,t). The adoption of deterministic routing rules on G_a makes possible the determination of relationships $\equiv_h$ at mapping time.
Furthermore, because messages can also interfere if their paths are disjoint (see Figure 1), for each channel (i,j), we define its *Interference Class* $S^h(i,j)$ as the equivalence class given from the transitive closure of relation $\equiv_h$. Classes $S^h(i,j)$ are a partition of all the messages exchanged during phase h. Only messages belonging to the same class may collide with each other during the considered communication phase.

Assuming equal for all sources the probability of obtaining a link, a generic message (i_h, j_h), $l_{i_hj_h}$ flits long, requiring the link simultaneously with $s-1$ others, of lengths $l_{i_1j_1}, \ldots, l_{i_{h-1}j_{h-1}}$, $l_{i_{h+1}j_{h+1}}, \ldots, l_{i_sj_s}$, is subjected to an approximate average delay of:

$$\frac{1}{2}\sum_{r\neq h} l_{i_rj_r}. \tag{8}$$

In fact (i_h, j_h) with probability $1/s$ will be the t-th message to travel on the link. In this case it will be blocked for the time required for the router to manage all $(t-1)$ preceding messages (each with probability $(t-1)(s-2)!/(s-1)! = (t-1)/(s-1)$ because each one can be in one of the $(t-1)$ positions before). The expected delay to which (i_h, j_h) will be subject is therefore:

$$\sum_{t=2}^{s}\frac{1}{s}\left(\sum_{r\neq h}\frac{t-1}{s-1}l_{i_rj_r}\right) = \frac{1}{s}\sum_{r\neq h} l_{i_rj_r}\frac{1}{s-1}\sum_{t=2}^{s}(t-1) =$$

$$= \frac{1}{s}\sum_{r\neq h} l_{i_rj_r}\frac{1}{s-1}\sum_{t=1}^{s-1} t = \frac{1}{s}\sum_{r\neq h} l_{i_rj_r}\frac{1}{s-1}\frac{(s-1)s}{2} =$$

$$= \frac{1}{2}\sum_{r\neq h} l_{i_rj_r}$$

Such analysis does not consider that routers, by always granting priority to the same sources, may not apply a fair policy.

From Eq. 8 and from the previous discussion, we can formulate a suitable approximation of the delays to which task j is subjected due to the blocking of messages delivered from processor i as:

$$R_{contention}(m,i,j) = \sum_{h=1}^{P} \sum_{\substack{(s,t)\neq(i,j) \\ (s,t)\in S^h(i,j)}} \frac{l_{st}}{2|S^h(i,j)|} \tag{9}$$

where $|S^h(i,j)|$ is the cardinality of Interference Class $S^h(i,j)$. As shown by Eq. 9, we evenly partition the delays expressed by Eq. 8 among all the messages belonging to the same Interference Class. This, because communication delays on different channels tend to overlap and not sum.

Adding Eq. 9 to Eq. 6, we arrive at our final estimate for $R(m,n)$, completion time of node n:

$$R(m,n) = R_{comp\&idle}(m,n) + \sum_{m(j)=n} R_{distance}(m,j)$$

$$- \sum_{m(j)=n} R_{multitasking}(m,j) + \sum_{m(i)\neq n, m(j)=n} R_{contention}(m,i,j) \tag{10}$$

Substituting Eq. 10 into Eq. 2, the formulation of our mapping evaluation function which measures program completion time in CPU cycles is completed.

4.1 Complexity and experimental evaluations

Computing Eq. 3 requires $\mathcal{O}(T)$, Eq.'s 4 and 5 $\mathcal{O}(na)$ together if we compute in $\mathcal{O}(n)$ the distance between two *pn*'s each time we need it.

The complexity of Eq. 9 depends strictly on some implementation details. Items in the same equivalence class are stored as nodes of a balanced tree whose root is the representative of the whole class [26]. The search for the representative of a class costs $\mathcal{O}(\log(a))$ and class merging takes $\mathcal{O}(1)$.

At the beginning, every item is the only member of its class. All messages are then routed (in $\mathcal{O}(akn)$ steps) on an appropriate data structure representing G_a, thus verifying (in $\mathcal{O}(\log a)$

steps) if a collision with a message in a different class may occur on each crossed link; if so, the two classes are merged together.

The previous sequence is repeated for each communication phase, with a global cost of $\mathcal{O}(Pkna \log a)$. At the end, Eq. 9 is computed in $\mathcal{O}(Pa \log a)$ steps at most. Eq. 9 -and the full evaluation function- takes therefore $\mathcal{O}(Pkna \log a)$.

In the best case, corresponding to entirely adjacent communicating tasks, the routing of a message requires $\Omega(1)$ and the complexity is therefore $\Omega(Pa)$.

Memory requirements are $\mathcal{O}(Pa)$ to store the p_{ij}^h array of bits, and $\mathcal{O}(a)$ for q_{ij}, l_{ij} and m_{ij} values. Moreover, $\mathcal{O}(PknN)$ words are required to hold the data structure representing G_a.

To measure the accuracy of the evaluation function F, 40 task graphs, composed of a number of nodes ranging from 4 to 200, have been randomly generated. On the average, each task interacts with seven partners and sends (or receives) a byte every eight processor cycles. Such high communication requirements, resulting in a large number of network collisions, have been chosen to render predictions of completion time more difficult. These graphs have been randomly mapped on a square mesh architecture with a number of processors not less than the number of tasks. For each graph both the value of function F and the actual completion time (obtained with a simulator [21]) have been computed. Corresponding values, sorted in increasing order of completion time, are showed in Figure 3. The maximum error of the evaluation function on these graphs is 5.8% and the average error only 1.7%. So we can trust that objective function F, when inserted in a greedy framework as the one proposed, gives a good approximation of what we really want to optimize.

5 The mapping algorithm

Mappers based on greedy algorithms may have polynomial costs because they search a unique path joining the root to a "good" leaf of the solution tree. In Figure 4 the pseudocode of the proposed mapping algorithm is reported. The program, starting from an empty mapping,

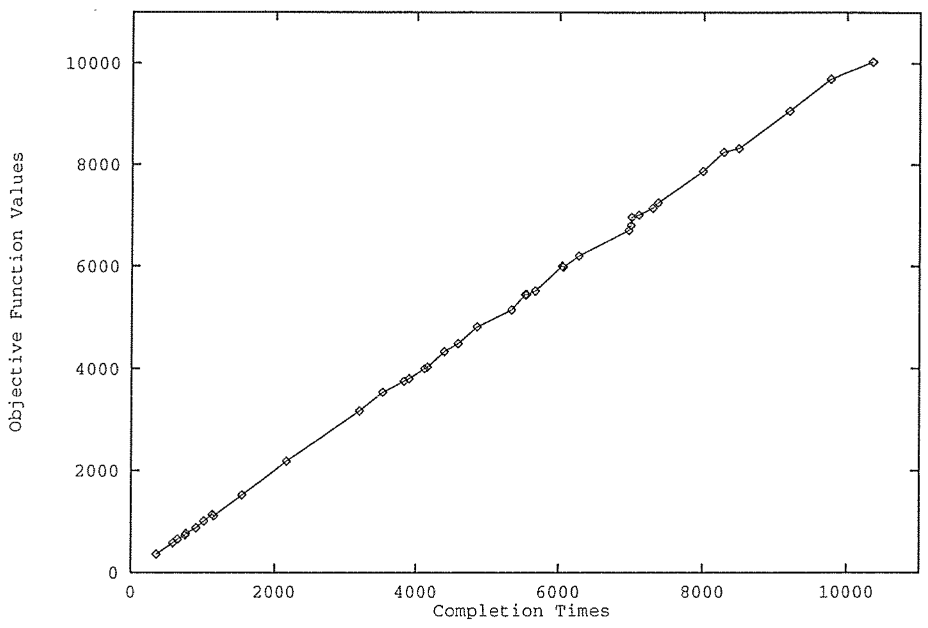

Figure 3: F estimates compared to actual completion time on random mappings.

repeatedly chooses a task from G_p and assigns it to a processor of G_a until all the tasks have been mapped or the solution reached becomes inadmissible (the implementation considers a mapping inadmissible when *pn*'s memory constraints are violated). Mapping inadmissibility causes failure of the algorithm that can be managed with backtracking. However, this problem should however arise very rarely because the accumulation of tasks on the same node usually produces growth in the value of the evaluation function.

First task and processor are chosen by a call to the *Select_First_Pair* function. This function returns the *centers* of G_p and G_a graphs, i.e. nodes that are at the minimum total distance from all others nodes ($i : \min_i \sum_j d(i,j)$).

The kernel of the mapping algorithm is function *Select_Pair* (see Figure 5) which takes a new task t from G_p by a call to *Select_Task*, and then chooses the best *pn* for executing t on the basis of values of the mapping evaluation function F.

```
Function Greedy_Mapper($G_p$ : $SPM\_description, G_a$ : $host\_graph$): $mapping$;
begin
    for $i := 1$ to $N$ $m(i) = \perp$;
    $(task, node) := Select\_First\_Pair(N_p, N_a)$;
    $N_p := N_p \setminus \{task\}$;
    $m(task) := node$;
    while $N_p \neq \emptyset$ do begin
        $(task, node) := Select\_Pair(N_p, N_a, m)$;
        $N_p := N_p \setminus \{task\}$;
        $m(task) := node$;
    end;
    Greedy_Mapper := $m$;
end;
```

Figure 4: Pseudocode of the greedy mapping algorithm.

```
Function Select_Pair($N_p$ : $task\_set$, $N_a$ : $node\_set$, $m$ : $mapping$) : $(t, n)$;
begin
    $task := Select\_Task(N_p)$;
    $Heap := \emptyset$;
    for $i := 1$ to $N$ begin
        $m(task) := i$;
        if $Feasible(m)$ then $Insert(Heap, i, F'(m))$;
    end;
    if $Count\_Elements(Heap) = 0$ then $Failure()$;
    $Rebuild(Heap)$;
    $i := Extract\_Node(Heap)$;
    $m(task) := i$;
    $m\prime := m$;
    for $j := 2$ to $\min(s, Count\_Elements(Heap))$ begin
        $i := Extract\_Node(Heap)$;
        $m\prime(task) := i$;
        if $F(m) > F(m\prime)$ then $m(task) := i$;
    end;
    Select_Pair := $(t, n)$;
end;
```

Figure 5: Pseudocode of the *Select_Pair* function.

5.1 Task selection strategy

Task selection policy, implemented within the function *Select_Task*, determines the order of G_p task assignment to the *pn*'s of G_a. Our choice has been to select next tasks on the basis of the presumed accuracy of the evaluation function they allow. Let $M = \{i \in N_p : m(i) \neq \perp\}$ be the set of tasks already mapped and $M' = \{i \in N_p : m(i) = \perp \wedge \exists j \in M \mid (i,j) \in E_p\}$ the set of unassigned tasks that interact with those in M.

If $M' \neq \emptyset$, function *Select_Task* returns the task t of M' that has the maximum number of partners in M. In fact, the more mapped partners of t there are, the more precisely the evaluation function can estimate completion time of the current partial mapping, driving the algorithm towards a good solution.

When this criterion is ambiguous because more tasks of M', say $t_1, \ldots, t_h$ have an equal number of partners in M, the *more connected* task t among $t_1, \ldots, t_h$ is chosen. To quantify the connectivity of t, tasks $i \in G_p$ with $d(t,i) < \min(D_a/4, D_p/4)$, where D_a and D_p are respectively the diameters of G_a and G_p, are counted. This second criterion is aimed at favoring the assignment of richly interconnected tasks when the availability of free *pn*'s is greater.

Otherwise, if $M' = \emptyset$ because G_p is disconnected, a randomly chosen task t is returned by function *Select_Task*. In this particular case communication and blocking times for selected task t cannot be forecast because where partners of t will be mapped cannot be foreseen.

5.2 Processor selection strategy

To improve efficiency, processor choice is performed in two steps (see code in Figure 5). Let t be the task returned by the last call to function *Select_Task*, m the current partial mapping and $C(j) = \{i \in M : \exists(i,j) \in E_p\}$ the set of partners of task j which are already assigned to some *pn*.

Firstly, a low–cost evaluation function F' is used to narrow the set of candidate *pn*'s. Values of function F' for mappings m', each obtained from m by assigning t to one of the admissible *pn*'s, are inserted into a heap that is subsequently ordered in increasing order of the F' field.

Estimate F' includes the computation of Eq. 6; furthermore, when a many-to-one mapping must be found because there are more tasks than processing nodes, F' is designed to favor *pn*'s to which some task has been already assigned if the *usage factor*, defined as the ratio between the number of already assigned tasks and the number of occupied *pn*'s, falls below the threshold $x = T/N$. This strategy avoids the need to assign last tasks of many–to–one mappings on busy *pn*'s, and experimentally yields better results (improvements of more than 5% on 50 many-to-one mappings of random graphs).

Secondly, the processor for task t is chosen on the basis of the evaluation function F among the first $s = (2 * n + 1) * |C(t)|$ *pn*'s present into the ordered heap.

Especially when only a few tasks have been mapped, it is possible for the evaluation function F to assume similar lower values for many of the s *pn*'s to which task t could be assigned. Let $j_1, \ldots, j_h$ (generically j) be the *pn*'s corresponding to similar values for evaluation function F. The choice among $j_1, \ldots, j_h$ is performed, in the order, on the basis of:

- the result of a look-ahead in the solution tree, making some optimistic assumptions about the mapping of some unassigned tasks: for each task $c \in M'$ partner of t, a task $c' \in C(c)$ is considered; assuming c to be mapped onto a *pn* x crossed by $path(j, m(c'))$ and/or by $path(m(c'), j)$, it is possible to estimate imprecisely -but efficiently- the overheads of j due to distance and contention. To this end $d(j, m(c'))$, the distance between j and $m(c')$, is added to the number of links of $path(j, m(c'))$ utilized by some channel of E_p (if $path(m(c'), j)$ crosses x as well, we also add the number of utilized links of $path(m(c'), j)$). An example (see Figure 6) should explain these details. Suppose that mapping task 1 of G_p on nodes j_1, j_2, j_3 of G_a yields the same value for the objective function F (the only messages for which the delay can be computed are those on channel (1,4)). Task 2 communicates with both task 1 and task 5 $\in M$. For all *pn*'s j_1, j_2, j_3, the distance from $m(5)$ is 2, but links of $path(j_2, 5)$ and of $path(5, j_2)$ are used by two different channels, while both paths between j_1 (or j_3) and 5 are free. So, j_1 or j_3 will be chosen;

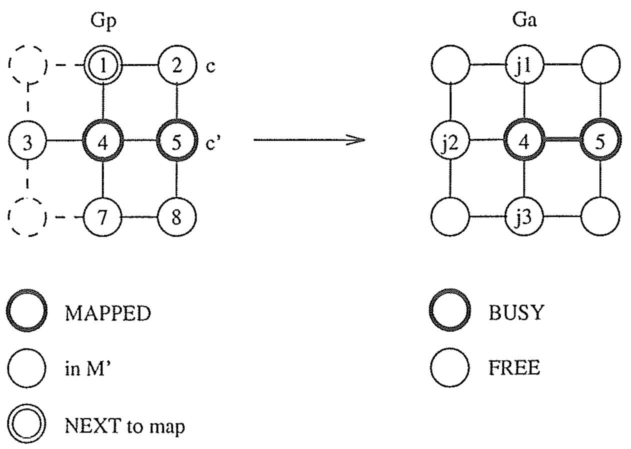

Figure 6: Look-ahead in the mapping of a 3x3 grid onto a isomorphic architecture.

- the values assumed by the *summed total cost* version of our mapping evaluation function: $F(m) = \sum_{n=1}^{N}(R(n))$;
- the absolute difference between the sum of the degrees of the neighbours of t in G_p and of j in G_a that have distance lower than g;
- the minimum n-dimensional rectangle holding all the mapped tasks.

5.3 Complexity and experimental performance

Select_First_Pair function requires $\mathcal{O}(n)$ and $\mathcal{O}(Ta)$ for computation of the centers of G_a and G_p respectively, while complexity of *Select_Pair* function, called exactly T times, is more difficult to state. Set M' is stored as a heap ordered on the basis of the two values used by function *Select_Task* to perform task choice. Each time a new task t is mapped, M' is updated inserting all its partners not yet assigned, and recomputing the two values used by *Select_Task*: $\mathcal{O}(T)$ steps are required to recompute the number of partners of t in M, while for the second value (task connectivity), whatever value for g is chosen, a time proportional to $\mathcal{O}(a)$ is needed. Being T the assignments, the previous steps take globally $\mathcal{O}(aT)$.

Moreover, updating M' each time a new task t is mapped requires at most the insertion or re–insertion of $D(t)$ partners (with $D(t)$ degree of task t). Thus, the global costs for managing M' is $\mathcal{O}(a \log T)$.

Task selection takes therefore globally $\mathcal{O}(aT)$.

The algorithm globally visits NT nodes of the solution tree. Function F' is computed for each visited node. F' requires $\mathcal{O}(D(i))$, with i the next task to map. Being the sum of the task degrees equal to $2a$, F' takes overall $\mathcal{O}(aN)$.

Rebuilding the heap of processors on the grounds of F' values is done T times, in $\mathcal{O}(TN)$ steps. Evaluation function F is computed s times at each level of the solution tree. At each level, we can derive F values from the value assumed on the previous partial mapping by storing the state of some data structures each time a new allocation is decided. In this way, we distribute the complexity of the F computation along the path from root to leaf of the solution tree. The costs of the other processor selection strategies are bounded by the F complexity. Thus, as $\mathcal{O}(Pkna \log a)$ is the complexity of F (see Section 4), a time of the order of $\mathcal{O}(sPkna \log a)$ it is required overall to choose processors.

Finally, global complexity in the worst case can be bounded by $\mathcal{O}(aT + aN + Ts \log T + Pskna \log a) \leq \mathcal{O}(Pla \log a)$ for every s. In the best case, corresponding to the absence of contention, complexity is $\Omega(Pla)$.

To measure the actual performance of the mapper, 40 graphs with up to 400 tasks, have been considered. One half of these graphs models mesh computations while the others, randomly generated, have high communication demands resulting, on the average, in 5.5 partners for each task and a ratio of 8 between processor cycles spent performing internal operations and flits sent or received. These programs were fed as input to the mapper choosing the smallest square mesh into which the tasks fit as the target architecture. Figure 7 reports the times (in seconds) needed on a DEC 3000/400 workstation to map each program as a function of the number of tasks. The curve labeled *mesh_graphs* (*random_graphs*) regards the execution times required to map graphs having mesh (random) communication pattern, while *WCC_random_graphs* and *BCC_random_graphs* (*BCC_random_graphs*) are the curves of theoretical Worst–Case and Best–

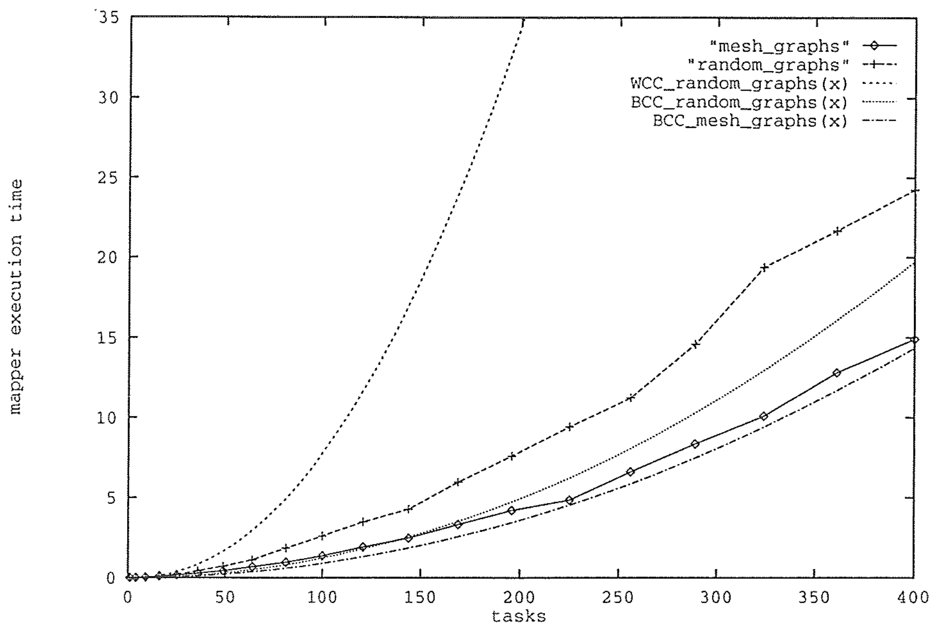

Figure 7: Mapping times on a DEC 3000/400 as a function of the number of input tasks.

Case complexities. Programs with 400 tasks require less than 25 seconds, and times for graphs with less than 225 tasks are shorter than 10 seconds. Moreover, the experimental times for random graphs are more close to the Best–Case than to the Worst–Case curve.

6 Results and concluding remarks

In order to verify the value of the proposed mapping algorithm, we report results achieved on three classes of experiments. The first class is composed of 6 graphs modelling real scientific programs. The second class of experiments considers graphs derived from highly parallel programs having a regular communication pattern for which the optimal mappings are known. The last trials compare our proposal with the results reported in Ref. [15].

6.1 Results on real programs

Six task graphs have been generated from the trace files distributed with the public domain *ParaGraph* package [12]. We automatically performed the translation by using a simple utility that reads *PICL* [11] timestamped event records that make-up the *ParaGraph* trace file, and then builds the corresponding TCG task graph.

The trace files used to generate the task graphs considered in this experiment are the following:

1. `chol.trf`: solution of a linear system by Cholesky factorization, forward solution, and back substitution. All–to–All communications on a 3–dimensional hypercube.

2. `fft64.trf`: typical dimensional exchange pattern of an FFT on a 6–dimensional hypercube.

3. `flops.trf`: program to count FLOPS on a 3–dimensional hypercube. All–to–All communications.

4. `matrix.trf`: matrix computations on a 4–dimensional hypercube. 2-D mesh communication pattern.

5. `sort.trf`: sorting algorithm on a 4–dimensional hypercube. Dimensional exchange communication pattern on a 4–dimensional hypercube.

6. `sparse.trf`: sparse Cholesky factorization. All–to–All communications on a 3–dimensional hypercube.

Table I reports the results achieved by the mapper on these real programs: the column labeled *Mapping* shows the assignments obtained running the mapper (i.e. tuple $(x_1, x_2, \ldots, x_T)$ means $m(i) = x_i, i = 1, \ldots, T$), while completion times of the resulting mappings are reported in the last column ($C.T.$). It is worth mentioning that task numeration was randomly permuted before passing the graphs as input to the mapper.

In all tests but the last, the obtained mappings are equivalent to the original ones (completion

Table I: Results on graphs derived from *ParaGraph* trace files.

Program	*Mapping*	*C.T.*
chol	(0,1,2,4,3,5,6,7)	178,787
fft64	(0,32,16,48,8,40,24,56,4,36,20,52,12,44,28,60, 2,34,18,50,10,42,26,58,6,38,22,54,14,46,30,62, 1,33,17,49,9,41,25,57,5,37,21,53,13,45,29,61, 3,35,19,51,11,43,27,59,7,39,23,55,15,47,31,63)	36,038
flops	(0,1,3,2,7,5,6,4)	304,391
matrix	(0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15)	199,802
sort	(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15)	69,070
sparse	(0,1,5,3,2,4,7,6)	472,261

times of the obtained mappings are exactly equal to that of the original mappings except on the `sparse` program, where we obtained a negligible improvement). On programs `fft64`, `matrix` and `sort` we are sure that the obtained mappings, as the originals, are optimal because they both exploit only nearest-neighbour communications (the mappings achieved on programs `fft64` and `matrix` are simply the bit reversal representation of the *pn*'s (tasks) binary identification numbers (i.e. $m((b_{n-1},\ldots,b_1,b_0)) = (b_0,b_1,\ldots,bn-1))$ and are therefore equal to the original ones because of the symmetry of hypercube topology).

6.2 Results on regular graphs

Table II reports the results achieved on TCG graphs of different sizes having regular communication patterns. As can be seen, the proposed mapping algorithm optimally embeds programs having *ring* and *2-D mesh* communication patterns onto mesh and hypercube multicomputers independent of the number of *pn*'s (more properly this independence holds true for all the tests performed). Graphs modelling *butterfly* communication patterns (e.g. a *2-D FFT* algorithm) are optimally embedded onto hypercubes while the resulting mappings onto mesh multicomput-

Table II: Results on regular graphs.

Comm. Pattern	*Target Topology*	*pn*'s	*Mapping*
`ring`	all k-ary n-cube	up to 512	optimal
`2-D mesh`	hypercube	up to 256	optimal
	2–D mesh	up to 784	optimal
`butterfly`	hypercube	up to 512	optimal
	2–D mesh	4,16,64,256	suboptimal
`tree`	2–D mesh	8,32,128,512	suboptimal

ers are nearly optimal (the tests we performed produced a maximum and average error of 1.8% and 0.5% with respect to the optimal mappings). Satisfactory are also the mappings of binary tree graphs modelling a reduce operation onto a *2–D mesh*. The obtained mappings produced a maximum and average error of 6.8% and 3.5% with respect to contention–free "H" mappings onto architectures with about twice the number of needed processors [13] .

6.3 Comparison with Lee–Aggarwal's results

In Ref. [15] the Synchronous Phases Model is proposed and an interesting mapping algorithm is described. The mapping algorithm is composed of two steps: in the first, an admissible mapping is obtained by execution of a greedy algorithm; the solution is then improved upon by executing a *pairwise exchange* algorithm with probabilistic jumps away from local optima. Results obtained with this mapper on a set of 9 graphs corresponding to real communication patterns, mapped on hypercubes of 8 and 16 nodes, are also reported in the paper.

The same graphs were fed as input to our mapper whose theoretical complexity is lower than Lee–Aggarwal's. Results of comparison are shown in Table III. Improvements, even if only slight at times, are obtained in 5 cases, equal completion times are achieved in 3 cases, only 1 result is worse.

Table III: Comparison with Lee-Aggarwal's results.

[15]	*Proposed Mapper*	
Time	*Time*	*Mapping*
66	66	(4 0 1 5 6 2 3 7)
64	64	(0 1 2 4 3 6 5 7)
93	86	(4 2 0 5 3 1 6 7)
136	**144**	(2 6 0 3 7 1 5 4)
102	98	(0 1 2 4 5 6 7 3)
67	67	(12 4 6 14 8 0 2 10 9 1 3 11 13 5 7 15)
99	98	(0 1 2 4 8 6 12 9 10 3 5 14 13 7 15 11)
107	97	(0 8 1 2 3 11 14 12 4 5 7 15 13 6 10 9)
159	158	(0 10 1 2 4 8 12 13 5 7 6 14 15 3 11 9)

6.4 Concluding remarks

Our investigation of the static task assignment problem has resulted in the development of a new greedy heuristic for the automated mapping of arbitrary unprecedence-constrained tasks onto highly parallel last generation multicomputers.

A $\mathcal{O}(Pkna \log a)$ objective function which efficiently approximates the completion time of a given program, modelled according to Lee–Aggarwal's abstraction, has been formulated as a measure of task assignment quality. Many experiments have demonstrated the accuracy of our objective function even when communication load is heavy or unevenly distributed over the network.

Based on this evaluation function, we have designed a greedy mapping algorithm. It adopts a novel look-ahead criterion which descends a level in the solution tree when the objective function assumes similar values for different assignments. The algorithm solves both topological and cardinality problems and can be target to deal with arbitrary multicomputers based on k-ary n-cube direct networks exploiting the *e-cube* routing algorithm and the *wormhole* flow control strategy.

Worst case complexity of the proposed mapping algorithm is bounded by $\mathcal{O}(Pla \log a)$, resulting in execution times shorter than 10 seconds on a workstation for highly communicating programs with less than 200 tasks. We have reported the performance of the proposed mapping algorithm for three classes of experiments: program–derived graphs with up to 64 tasks, regular graphs with up to 784 tasks, and Lee–Aggarwal's graphs. In all cases the obtained assignments are optimal or reasonably good, showing the suitability and effectiveness of the followed approach when program behavior is loosely dependent on input values.

References

[1] A. Agarwal. Limits on Interconnection Network Performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–411, October 1991.

[2] S. Antonelli, F. Baiardi, S. Pelagatti, and M. Vanneschi. A Static Approach to Process Mapping in Massively Parallel Systems. In *Proceedings of IFIP Working Conf. on Parallel Processing*, Pisa, Italy, April 1988.

[3] S. Antonelli, F. Baiardi, S. Pelagatti, and M. Vanneschi. Communication Cost and Process Mapping in Massively Parallel System: a Static Approach. Technical Report TR 12/89, Dept. of Computer Science, University of Pisa - Italy, 1989.

[4] S. Arunkumar and T. Chockalingam. Randomized Heuristics for the Mapping Problem. *Journal of High Speed Computing*, 4(4):289–299, April 1992.

[5] T. Baba, Y. Iwamoto, and T. Yoshinaga. A Network-Topology Independent Task Allocation Strategy for Parallel Computers. In *Proceedings of Supercomputing*, 1990.

[6] F. Berman and L. Snyder. On Mapping Parallel Algorithms into Parallel Architectures. *Journal of Parallel and Distributed Computing*, 4(5):439–458, October 1987.

[7] S. H. Bokhari. On the Mapping Problem. *IEEE Transactions on Computers*, 30(3):207–214, March 1981.

[8] S. W. Bollinger and S. F. Midkiff. Heuristic Technique for Processor and Link Assignment in Multicomputers. *IEEE Transaction on Computers*, 40(3):325–333, March 1991.

[9] T. L. Casavant and J. G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transaction on Software Engineering*, 14(2):141–154, February 1988.

[10] W. Dally. Network and Processors Architecture for Message-Driven Computers. In R. Suaya and G. Bithwistle, editors, *VLSI and Parallel Computation*, chapter 3, pages 140–222. Morgan Kaufmann Publisher, Inc. - San Mateo, California, 1990.

[11] G. A. Geist et al. A User's Guide to PICL: a Portable Instrumented Communication Library. Technical Report TM-11616, Oak Ridge National Laboratory, September 1990.

[12] M. T. Heath and J. E. Finger. ParaGraph: A Tool for Visualizing Performance of Parallel Programs. ParaGraph user guide, available from `netlib.ornl.gov`, May 1993.

[13] E. Horowitz and A. Zorat. The Binary Tree Interconnection Network: Application to Multiprocessor System and VLSI. *IEEE Transactions on Computers*, C-30:247–253, April 1981.

[14] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[15] S. Y. Lee and J. K. Aggarwal. A Mapping Strategy for Parallel Processing. *IEEE Transactions on Computers*, 36(4):433–442, April 1987.

[16] V. M. Lo. Heuristic Algorithms for Task Assignment in Distributed Systems. *IEEE Transactions on Computers*, 37(11):1384–1397, November 1988.

[17] V. M. Lo. Temporal Communication Graphs: Lamport's Process-Time Graphs Augmented for the Purpose of Mapping and Scheduling. *Journal of Parallel and Distributed Computing*, 16(4):378–384, December 1992.

[18] P. Y. R. Ma, E. Y. S. Lee, and M. Tsuchiya. A Task Allocation Model for Distributed Computing Systems. *IEEE Transactions on Computers*, 31(1):41–47, January 1982.

[19] H. Mülenbein, M. Gorges-Schleuter, and O. Krämer. New Solutions to the Mapping Problem of Parallel Systems: the Evolution Approach. *Parallel Computing*, 4(3):269–279, June 1987.

[20] M. G. Norman and P. Thanisch. Models of Machines and Computation for Mapping in Multicomputers. *ACM Computing Surveys*, 25(3):263–302, September 1993.

[21] S. Orlando, R. Perego, and M. Sardo. PNS, a Programmable Simulator of Direct Networks. Technical Report R/08/44, C.N.R. Finalized Project "Sistemi Informatici e Calcolo Parallelo", 1993.

[22] P. Sadayappan, F. Ercal, and J. Ramanujam. Cluster Partitioning Approaches to Mapping Parallel Programs onto a Hypercube. *Parallel Computing*, 13(1):1–16, January 1990.

[23] C. C. Shen and W. H. Tsai. A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion. *IEEE Transactions on Computers*, 34(3):197–203, March 1985.

[24] J. B. Sinclair. Efficient Computation of Optimal Assignments for Distributed Tasks. *Journal of Parallel and Distributed Computing*, 4(2):342–362, April 1987.

[25] H. S. Stone. Multiprocessor Scheduling with the aid of Network Flow Algorithms. *IEEE Transactions on Software Engineering*, 3(1):85–93, January 1977.

[26] R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.