Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs*

François Pellegrini
LaBRI, URA CNRS 1304
Université Bordeaux I
351 Cours de la Libération, 33405 TALENCE CEDEX, FRANCE
pelegrin@labri.u-bordeaux.fr

Abstract

The combinatorial optimization problem of assigning the communicating processes of a parallel program onto a parallel machine so as to minimize its overall execution time is referred to as static mapping. *This problem is NP-complete in general.*
In this paper, we introduce a mapping algorithm based on the recursive bipartitioning of both the source process graph and the target architecture graph, whose divide & conquer *and modular approach allows the handling of many topologies and bipartitioning heuristics. Mapping results on hypercube, binary de Buijn, and bidimensional mesh graphs are presented in order to illustrate this feature.*

1 Introduction

The efficient execution of a parallel program on a parallel machine requires that the communicating processes of the program be placed so as to minimize its overall execution time. This operation, referred to as *mapping*, can be either *static*, if the mapping occurs prior to the execution of the program and is never modified, or *dynamic*, when processes may be moved between processors at run time.
Static mapping has been proven to be NP-complete in the general case [9]. Therefore, many studies have been carried out in order to find sub-optimal solutions in reasonable time [3, 4, 7, 13, 14, 15].

In this paper, we present a mapping algorithm, based on the recursive bipartitioning of both the graph of processes and the graph that represents the parallel machine, which proceeds by recursive allocation of subsets of processes to subsets of processors.

After some important definitions in section 2, we give in section 3 the frame of our algorithm. Section 4 is devoted to graph bipartitioning algorithms, and particularly to our extension of the Fiduccia-Mattheyses algorithm. Some experimental results are then given in section 5 to show the speed and efficiency of our mapper, after which we discuss parallel aspects of our algorithms.

2 Static mapping and cost functions

2.1 Standard definitions

The computation graph which represents the parallel program to map onto the target architecture, also called the *source graph* S, is a valuated undirected graph whose vertices represent the processes of the parallel program, and whose edges represent communication channels between communicating processes. Vertex- and edge- valuations associate with every vertex v_S and every edge e_S of S integer numbers $c_S(v_S)$ and $c_S(e_S)$ which represent estimations of the computation cost of the corresponding process and the amount of communication to be transmitted on the channel, respectively.

The architecture graph, also referred to as the *target graph* T, represents the topology of the target machine. We assume that the machine is homogeneous, so the target graph is not valuated.

A *mapping* from S to T consists of two applications $\varphi : V(S) \longrightarrow V(T)$ and $\psi : E(S) \longrightarrow \{E(T)\}$, such that $\varphi(v_S) = v_T$ if process v_S is mapped onto processor v_T, and $\psi(e_S) = \{e_T^1, e_T^2, \ldots, e_T^n\}$ if communication channel e_S is routed through communication

*This work was supported by the French GDR C^3

links e_T^1, e_T^2, ..., e_T^n. $|\psi(e_S)|$ denotes the dilation of edge e_S, *i.e.* the number of edges of $E(T)$ used to route edge e_S.

2.2 Cost functions

The goal of a mapping is to minimize the overall execution time of a parallel program on the parallel machine onto which it is mapped. The prediction of the quality of a given mapping with respect to this performance goal is usually achieved by means of a *cost function*, which may account for parameters such as load balancing on the target processors, communication load balancing on communication links, minimization of inter-processor communication, minimization of the dilation of the source graph edges, *etc*.

Static mapping heuristics are algorithms designed to find mappings which minimize this cost function. However, due to the variety of parallel machines, it is difficult to compute a unified cost function. In particular, the trade-off between computation load balance and communication cost is hard to tune, and depends on the technologies used in these machines.

For given source and target graphs, the average overall computation load to be distributed can be computed, as opposed to the amount of inter-processor communication for which no estimation is available. Thus, we use a cost function whose goal is to minimize communication cost on the target links while keeping the load balance within a predefined tolerance. Several graph bipartitioning methods, such as Kernighan-Lin [11], Fiduccia-Mattheyses [8], and Pothen-Simon-Liou [16], use this approach; for others, such as genetic algorithms, the balance constraint must be accounted for as a strong penalty term in a unique cost function, as it is done in [6].

The minimization of our cost function implies that intensively inter-communicating processes should be mapped onto as close to each other as possible processors. Therefore, when given some bipartition of the target architecture, we tend to prefer mappings which minimize the resulting *edge cut function* f_c, which accounts for communication costs as well as edge dilation:

$$f_c(\varphi, \psi) = \sum_{e_S \in \mathrm{cut}(E(S))} (c(e_S) \cdot |\psi(e_S)|) \ ,$$

where $\mathrm{cut}(E(S))$ represents the set of the edges of $E(S)$ whose ends are mapped to processors belonging to the two subsets of the bipartition.

3 Frame of the mapping algorithm

3.1 Sketch of the algorithm

The method we describe is based on a *divide & conquer* approach. It proceeds by recursive allocation of subsets of processes to subsets of processors, till the processor subsets are restricted to one element or the process subsets are empty. At each step, the algorithm performs the bipartitioning of the subset of processors, also called *domain*, into two disjoint subdomains, and calls a *graph bipartitioning* algorithm to map the subset of processes onto the two subdomains, with respect to our cost function, as written in the following sketch.

```
mapping (D, P)
Set_Of_Processors  D;
Set_Of_Processes   P;
{
  Set_Of_Processors  D0, D1;
  Set_Of_Processes   P0, P1;

  if (|P| == 0) return;  /* If nothing to do.     */
  if (|D| == 1) {        /* If one processor in D */
    result (D, P);       /* P is mapped onto it.  */
    return;
  }

  (D0, D1) = processor_bipartition (D);
  (P0, P1) = process_bipartition   (P, D0, D1);
  mapping (D0, P0);      /* Perform recursion. */
  mapping (D1, P1);
}
```

The above algorithm relies on the ability to define four main objects:

- A *domain* structure, which represents a set of processors.
- A *domain bipartitioning function*, which, given a domain, bipartitions it into two disjoint subdomains.
- A *process bipartitioning function*, which, given a domain and its two subdomains, and a process set, bipartitions it into two disjoint process sets to be mapped onto each subdomain. Process bipartitioning functions are studied in section 4.
- A *domain distance function*, which gives, in the target graph, a measure of the distance between two disjoint domains. Since domains may not be convex nor connected, this distance may be estimated. However, it must respect certain homogeneity properties, such as giving more accurate results as domain sizes diminish.
  The domain distance function is used in the process bipartitioning algorithms to compute the

communication function to minimize, since it allows us to estimate the dilation of the edges linking vertices which belong to different domains.
Using such a distance function amounts to considering in our computations that all routings will use shortest paths on the target architecture. This is not unreasonable to assume, as new generations of parallel machines tend to handle routing dynamically with shortest-path routings (for instance, the Intel Paragon uses Manhattan-style routing on its bidimensional mesh topology). We have thus chosen that our program would not provide routings for the communication channels, leaving their handling to the communication system of the target machine.

All these routines are seen as black-boxes by the mapping program, which can thus accept any kind of target architecture and process bipartitioning function. As a matter of fact, according to the topology of the target architecture, the results of the domain functions can be algorithmically computed at run-time, or be extracted from precomputed tables, which allows us to use exotic topologies such as Butterfly and de Bruijn graphs [12].

For instance, for the two-dimensional mesh target architecture, we have implemented domains as rectangular areas, the domain bipartitioning function as the function which splits a domain along its smallest dimension into two parts of equivalent sizes (within one row or column), and the distance function as the smallest distance between the centers of the two domains.
For the hypercube target architecture, domains are sub-hypercubes, the domain bipartitioning function splits a hypercube into two sub-hypercubes, and the domain distance function returns the number of different bits in the labeling of sub-hypercubes.
The domain decomposition of binary de Bruijn graphs lies on the fact that an undirected binary de Bruijn graph of diameter D, denoted $UB(2, D)$, contains two copies of a spanning subgraph of $UB(2, D-1)$. The leftmost bit of any D-digit vertex label word indicates the copy to which it belongs, and the $(D-1)$-digit label of the vertex within this copy is built by xor-ing pairwise the D digits of the original label. For example, vertex 01100 of $UB(2,5)$ belongs to the first instance of subgraph of UB(2,4), within which it has label 1010. This decomposition is recursively applied to build the graph decomposition used by our algorithm.

3.2 Execution scheme

From an algorithmic point of view, our mapper behaves as a greedy algorithm (the mapping of a process to a subdomain is definitive), at each step of which iterative algorithms can be applied.

The double recursive call performed at each step induces a recursion scheme in the shape of a binary tree, each vertex of which corresponds to a bipartitioning job, *i.e.* the bipartitioning of both the current domain and process set.
In the case of a depth-first sequencing, as programmed in the above sketch, bipartitioning jobs called in the left branches have no information on the distance of vertices to be processed by the right branches.
On the contrary, sequencing the jobs according to a by-level (breadth-first) travel of the tree allows that, at any level, any bipartitioning job may have information on the subdomains to which all the processes have been allocated during the previous level. Thus, when deciding in which subdomain to put a given process, a bipartitioning job can account for the communication costs induced by the neighbor processes, whether they are handled by the job itself or not, since it can estimate the dilation of the corresponding edges. This results in an interesting feed-back effect: once an edge has been kept in a cut between two subdomains, the distance between its end vertices will be accounted for in the communication cost function to be minimized, and following jobs will thus be more likely to keep these vertices close to each other, as illustrated in figure 1.

Moreover, since all domains are split at each level, they all have equivalent sizes, which respects the distance homogeneity and gives the algorithm more coherence.
For all what precedes, we have implemented the second approach in our mapper: bipartitioning jobs are stored into a queue, the execution of a job resulting in the creation of at most two new jobs, which are in turn enqueued.

One can notice that, by using the hypercube as target topology and depth-first execution, our mapping program is identical in behavior to the one of [7]. In that sense, our work, by formalizing the concepts of domain, distance, and execution scheme, can be seen as a generalization of their work which handles many target topologies and graph bipartitioning methods.

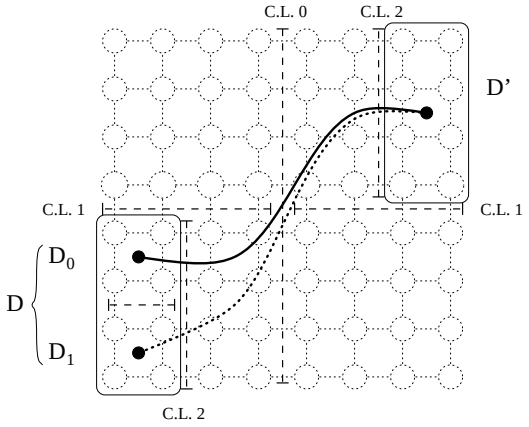

Figure 1: The job which maps the processes of domain D to subdomains D_0 and D_1 accounts in its communication cost function for the estimated distance from vertices which were mapped to D' at the previous level (C.L. stands for *Cut Level*).

4 Process bipartitioning methods

4.1 General considerations

In the by-level implementation of the mapping algorithm, bipartitioning jobs are fed with the set of processes to bipartition, the two subdomains into which to put them, and the domain mapping computed at the previous level for all the source processes. On completion, they return two subsets of processes, one for each subdomain.

The recursive mapping algorithm uses process bipartitioning methods as black boxes. It allows us to run any type of graph bipartitioning method which accepts our cost function, such as Kernighan-Lin-type algorithms [11], simulated annealing [4], quadratic assignment [17], genetic algorithms [15, and included references], *etc*. Moreover, *mapping strategies* can be defined by letting each job select the graph bipartitioning method that it will use, according to local properties of its input (size of the mapping domain, distribution of the processes weights, *etc*).

4.2 An improvement of the Fiduccia-Mattheyses heuristics

The first bipartitioning method that we have implemented is an improvement of the Fiduccia-Mattheyses algorithm which enables us to handle huge gains and non-unity vertex loads. The Fiduccia-Mattheyses graph bipartitioning heuristics [8] is an almost-linear improvement of the Kernighan-Lin algorithm [11]. Its goal is to minimize the cut between two vertex subsets, while maintaining the balance of their cardinals within a limited user-specified tolerance. Starting from an initially balanced solution of any cut value, it proceeds iteratively by trying, at each stage, to reduce the cut of the current solution.

The algorithm maintains, for all the vertices of both subsets, a *gain* value, which represents the value by which the current cut would decrease if the vertex were moved to the other subset (gains may thus be negative). Vertices of identical gain are linked into *linked gain lists* which are stored as entries of a *gain array*.

At each stage, the algorithm builds an *ordering* of as many vertices as it can, by repeating the following process: it chooses from the two subsets the vertex with the highest current gain (which may be negative) and whose move will not set the balance out of the user-specified tolerance. When a vertex is chosen, the algorithm pretends to move it and updates the gains of its neighboring vertices accordingly. This is repeated till all vertices have been chosen or no move of yet unprocessed vertices would keep the load balance. Once the ordering is complete, the new solution is built from the current one by moving as many vertices of the ordering as necessary to get the maximum accumulated gain. Thus, by considering the accumulated gain, the algorithm allows *hill-climbing from local minima* of the cut cost function.

The almost-linearity of this algorithm is based on two limitations. First, it is supposed that the vertex degrees are small and all edges have unity weight, so the range of the gain values is small, and thus the search in the gain array for vertices of best gain is assumed to take an almost-constant time. Second, all vertices are supposed to have equal (unity) weights, so moving the head of a given gain list is equivalent in balance to moving any vertex in this list.

Unfortunately, the above is no longer true when vertices and edges have non-unity weights and when the gain accounts for the distance between the edge ends.
First, it results in huge vertex gains, whose handling requires new data structures. We have tried several methods, and have consequently chosen to index the gain table with a logarithmic scale, assuming that vertices whose real gains differ at most by a factor two would have almost equivalent behaviors if they were moved.
Second, because of the non-unity vertex gains, mov-

ing the head of a gain list may not result in the best load balance, compared to other vertices in the same list. As no data structure could easily avoid the evaluation of vertices whose load would cause imbalance if they were moved, and since we did not want to scan the whole lists for the vertex of best load balance, we have chosen to scan the gain list of best logarithmic gain till a vertex whose move would not cause imbalance is found.

4.3 Handling of imbalanced graphs

The handling of imbalanced process graphs, *i.e.* graphs within which some processes have weights much bigger than the average, leads to several problems.

Let us consider for instance a complete process graph such that all vertices have weight one, except for one sole vertex whose weight is equal to the number of vertices of the graph. If the bipartitioning algorithm were run on this graph, the heaviest vertex would be put in one subdomain, and all the others in the other, leaving all processors of the first subdomain idle, except for only one.

To prevent this behavior, we have implemented an adaptative weight limitation procedure. At the beginning of every job, it computes the *effective weights* of all the processes, which are the minimum of the *real weights* of the processes and a multiple of the average real weight of the processes mapped to the current domain. The multiplicative factor, which is always greater than one, evolves with respect to domain size so that effective weights tend to match real weights as domain size diminishes. The use of effective weights by the bipartitioning algorithms amounts, in the first levels, to computing the load balance more in terms of process numbers than in terms of process loads, which reduces the impact of pathological cases such as the one described above.

5 Performance evaluations

5.1 Test graphs

Our program has been tested with several kinds of source graphs, two of which are presented here.

We first studied the valuated interprocess communication graphs issued from a parallel implementation of a sparse block Cholesky factorization solver. Process graph *rcInit* represents the partition of the unknowns of a matrix induced by a nested dissection method. Process graph *rcRefn* is obtained from *rcInit* by a graph refinement phase, where heavily loaded vertices are split into subgraphs of lightly loaded vertices, in order to obtain a better granularity of the problem, at the expense of vertex and edge creations. See [5] for a complete description of the method.

A second class of test graphs, obtained from [10], represents finite-element-type triangular and quadrilateral grids, linked to problems of fluid dynamics or structural mechanics. Process graph *ha4elt* is a decomposition of the area around a multi-elements airfoil, and *haBrack* is the geometrical description of a bracket. These two graphs have unity vertex and edge loads.

5.2 Performance criteria

The quality of our mappings is evaluated with several parameters, whose expressions are given below. $\min_{map}$, $\max_{map}$, and μ_{map} are the minimum, maximum, and average process loads mapped onto target processors, respectively. μ_{dil} represents the average dilation of the source graph channels with respect to the target topology, μ_{exp} the average cost-weighted dilation (both of them are computed by the means of shortest paths), and μ_{com} the average communication load of the source graph.

$$\mu_{dil} = \frac{\sum_{e_S \in E(S)} |\psi(e_S)|}{|E(S)|} \qquad \mu_{exp} = \frac{\sum_{e_S \in E(S)} (c(e_S) \cdot |\psi(e_S)|)}{\sum_{e_S \in E(S)} c(e_S)}$$

$$\mu_{com} = \frac{\sum_{e_S \in E(S)} c(e_S)}{|E(S)|}$$

In addition to the above, we have defined the synthetic parameters ε_{map} and ε_{exp}.

$$\varepsilon_{map} = 1 - \frac{\sum_{v_T \in V(T)} |c(v_T) - \mu_{map}|}{\mu_{map} \cdot |V(T)|}$$

$$\varepsilon_{exp} = 1 - \frac{\mu_{exp}}{\mu_{dil}}$$

ε_{map}, in the range $[1; -\infty[$, is equal to 1 if the mapping is ideally balanced, and tends towards negative numbers when the mapping gets imbalanced. ε_{exp}, in the range $[1; -\infty[$, is positive if the mapping strategy has succeded into putting heavily communicating processes closer to each other than it has done for lightly communicating processes; it is therefore equal to 0 if all edges have same weights.

5.3 Mapping results

All the results presented were obtained with our mapper being configured so that all the bipartition-

Target	H(8)			
Graph	rcInit	rcRefn	ha4elt	haBrack
Vertices	2047	3470	15606	62631
Edges	7750	135148	45878	366559
$\min_{map}$	6544	58380	58	234
$\max_{map}$	686298	61332	64	252
μ_{map}	59639	59639	61	245
μ_{dil}	2.265	2.915	0.347	0.485
μ_{exp}	2.553	2.846	0.347	0.485
ε_{map}	0.004	0.992	0.987	0.990
ε_{exp}	-0.127	0.024	0.000	0.000
Sequential time	11.70	93.80	138.09	1069.84

Table 1: Mapping results on H(8).

Target	UB(2,8)			
Graph	rcInit	rcRefn	ha4elt	haBrack
Vertices	2047	3470	15606	62631
Edges	7750	135148	45878	366559
$\min_{map}$	7178	57812	58	237
$\max_{map}$	686298	61628	64	251
μ_{map}	59639	59639	61	245
μ_{dil}	3.483	4.613	0.622	0.715
μ_{exp}	3.670	4.581	0.622	0.715
ε_{map}	0.004	0.991	0.986	0.991
ε_{exp}	-0.053	0.007	0.000	0.000
Sequential time	12.17	93.60	137.34	1138.44

Table 2: Mapping results on UB(2, 8).

ing jobs ran our Improved Fiduccia-Mattheyses graph bipartitioning method, with the load balance tolerance of the algorithm being set, for every job, to $0.005 \times \max($*ideal load average on domain processors, load of effective heaviest process in subset*$)$. The test machine is a SUN 4/460 with 32 Mb of main memory and 64 Mb of disk swap.
The test graphs described above were all mapped on a hypercube of dimension 8, denoted $H(8)$, a binary de Bruijn graph of diameter 8, and a 16×16 bidimensional mesh, which all have 256 nodes. Results are given in tables 1, 2, and 3, with all user times in seconds.

Pleasantly, it seems from all our experiments that

Target	M2(16,16)			
Graph	rcInit	rcRefn	ha4elt	haBrack
Vertices	2047	3470	15606	62631
Edges	7750	135148	45878	366559
$\min_{map}$	211	57969	58	237
$\max_{map}$	686298	61748	64	253
μ_{map}	59639	59639	61	245
μ_{dil}	4.574	5.611	0.606	0.819
μ_{exp}	5.782	5.462	0.606	0.819
ε_{map}	0.004	0.991	0.987	0.990
ε_{exp}	-0.264	0.027	0.000	0.000
Sequential time	11.87	91.06	134.14	1083.89

Table 3: Mapping results on M2(16, 16).

the mapper has an almost-linear behavior with respect to the number of channels of the source graphs, even for graphs with non-unity vertex and edge loads.

At constant number of processors, the density and the diameter of the target graphs have a direct impact on the ε_{map} and ε_{exp} coefficients: their values for the unbounded-degree hypercube are smaller that those for the bounded-degree de Bruijn and mesh graphs, and the values for the de Bruijn graph of diameter 8 are smaller than the ones for the mesh of diameter 30.

As one might expect, the mapper handled the almost-balanced *rcRefn* graph much better than the imbalanced *rcInit* (whose heaviest *process* load is 686298), which is shown by ε_{map} and ε_{exp}. The results issued by our mapper for all graphs of this kind are equivalent to the ones computed with a two-phase Bokhari algorithm, but in a time orders of magnitude smaller: a few tens of seconds compared to several tens of minutes (see [5]).

Results for the *ha** graphs show the ability of the mapper to handle large graphs, providing good mappings in a reasonable time.
In [10, table 3.2] are summarized mapping results obtained for a hypercube of diameter 8 with the Cyclic Pairwise Exchange heuristics, which allow us to compute the equivalent μ_{exp} parameter for these mappings. It is equal to 0.526 for the *ha4elt* case, and 0.489 for the *haBrack* case, which are slightly greater than the 0.347 and 0.485 values obtained with our Dual Recursive Bipartitioning algorithm. The average edge expansion of our mappings is thus slightly better than the one obtained by CPE. In the same way, our dilation distributions are also slightly better, since more processes are placed at smaller distances from their neighbors by our DRB than by the CPE heuristics.

6 Parallel aspects of the method

Although the mapping program runs sequentially at this time, its divide & conquer approach and its design make it suitable for parallel execution. In particular, the by-level sequencing of the bipartitioning jobs tree has several interesting consequences.

First, it requires a by-level synchronization of the jobs, which is compatible with parallelism: if the jobs of a same level are processed on different host processors, the synchronization amounts to a *gossiping* phase during which host processors inform each other of the

subdomain allocation they have performed. These data will be used by the jobs of the next level to compute the communication cost function.

Second, the number of bipartitioning jobs doubles at each level. A good host topology for off-line execution of the mapper could thus be the hypercube, as there is a direct link, on yet unused dimensions, between any active processor and the processor used in the next job level to run the second subjob it has computed.
However, this approach does not use the host machine efficiently, since the first (and biggest) bipartitioning job runs on a single processor. Here appears the interest of mapping strategies: first jobs can be handled with parallel graph bipartioning methods distributed on many, unused yet, host processors (such as parallel versions of genetic algorithms [15] and simulated annealing [2, and included references]), till jobs become numerous and small enough to be distributed and run sequentially on distinct host processors.

Last, if the host machine on which the mapping is run is the target machine, then every host processor gets at the last level the list of the processes that are mapped onto it, and can load and run them, resulting in an on-line mapper and loader.

7 Conclusion

In this paper, we have described the frame of a static mapper using dual recursive bipartitioning of both the process and architecture graphs which, thanks to its divide & conquer approach, allows parallelization. Several experimental results have been presented, which show its efficiency.

Current work includes the porting of the code to the Intel Paragon parallel machine, as well as its integration into the ADAM programming environment [1]. The mapper is also being used to quickly compute mappings of the data partitions of matrices, which serve as initial data distributions for the dynamic load balancer of the sparse parallel block Cholesky solver being developed at the LaBRI.

Acknowledgements

I thank Jean Roman very much for his thoughtful comments and remarks. Also, many thanks to Johny Bond for the de Bruijn decomposition method, to Steve Warren Hammond who kindly gave me his test graphs, and to Robert Strandh who carefully read this article and purified its English.

References

[1] M. Alabau, S. Chaumette, M.-C. Counilh, J.-M. Lépine, J. Roman, and B. Vauquelin. *A programming environment dedicated to a model of explicit parallelism*, volume 6 of *Advances in parallel computing*, pages 193–212. North-Holland, 1993.

[2] R. Azencott. *Simulated Annealing, Parallelization Techniques.* 1992.

[3] S. H. Bokhari. On the mapping problem. *IEEE Transactions on Computing*, C-30(3):207–214, 1981.

[4] S. W. Bollinger and S. F. Midkiff. Processor and link assignment in multicomputers using simulated annealing. In *Proceedings of the 11th Int. Conf. on Parallel Processing*, pages 1–7. The Penn. State Univ. Press, aug 1988.

[5] P. Charrier and J. Roman. Partitioning and mapping for parallel nested dissection on distributed memory architectures. In *LNCS 634 – Proceedings of CONPAR'92*, pages 295–306. Springer-Verlag, sep 1992.

[6] T. Chockalingam and S. Arunkumar. A randomized heuristics for the mapping problem: The genetic approach. *Parallel Computing*, 18:1157–1165, 1992.

[7] F. Ercal, J. Ramanujam, and P. Sadayappan. Task allocation onto a hypercube by recursive mincut bipartitionning. *Journal of Parallel and Distributed Computing*, 10:35–44, 1990.

[8] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, pages 175–181. IEEE, 1982.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractablility: A Guide to the Theory of NP-completeness.* W. H. Freeman, San Francisco, 1979.

[10] S. W. Hammond. *Mapping unstructured grid computations to massively parallel computers.* PhD thesis, Research Institute for Advanced Computer Science, jun 1992.

[11] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitionning graphs. *BELL System Technical Journal*, pages 291–307, feb 1970.

[12] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: arrays, trees, hypercubes*. Morgan Kaufman Publisher, 1992.

[13] V. M. Lo. Heuristic algorithms for task assignment in distributed systems. In *International Conference on Distributed Computer Systems*, pages 30–39. IEEE, 1984.

[14] P. R. Ma, E. Y. S. Lee, and M. Tsuchiya. A task allocation model for distributed computing systems. *IEEE Transactions on Computers*, C-31(1):41–47, jan 1982.

[15] T. Muntean and E.-G. Talbi. A parallel genetic algorithm for process-processors mapping. *High performance computing*, 2:71–82, 1991.

[16] A. Pothen, H. D. Simon, and K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal of Matrix Analysis*, 11(3):430–452, jul 1990.

[17] C. Roucairol and P. Hansen. Cut cost minimization in graph partitioning. *Numerical and Applied Mathematics*, pages 585–587, 1989.