Task Allocation onto a Hypercube by Recursive Mincut Bipartitioning

F. Ercal[*], J. Ramanujam and P. Sadayappan[†]
Department of Computer and Information Science
The Ohio State University
Columbus Ohio 43210

September 1, 1989

[*]Current address: Department of Computer Engineering and Information Science, Bilkent University, Ankara, Turkey.
[†]This work was supported in part by an Air Force DOD-SBIR program through Universal Energy Systems, Inc. (S-776-000-001) and by the State of Ohio through the Thomas Alva Edison Program (EES-529769).

Proposed running head: Task Allocation by Recursive Partitioning

Name and mailing address of author to whom correspondence should be sent:

P. Sadayappan
Department of Computer and Information Science
The Ohio State University
2036 Neil Avenue
Columbus OH 43210-1277

Abstract

An efficient recursive task allocation scheme, based on the Kernighan-Lin mincut bisection heuristic, is proposed for the effective mapping of tasks of a parallel program onto a hypercube parallel computer. It is evaluated by comparison with an adaptive, scaled simulated annealing method. The recursive allocation scheme is shown to be effective on a number of large test task graphs – its solution quality is nearly as good as that produced by simulated annealing, and its computation time is several orders of magnitude less.

List of Symbols used:

V	Italic letter (capital) "vee"
E	Italic letter (capital) "ee"
k^{th}	Italic "kay" superscript "th"
N	Italic letter (capital) "en"
i	Italic letter (lower case) "eye"
j	Italic letter (lower case) "jay"
w_i	Italic "double u" subscript "eye"
c_{ij}	Italic "see" subscript "eye" and "jay"
P	Italic letter (capital) "pee"
E_p	Italic letter (capital) "ee" subscript "pee"
K	Italic letter (capital) "kay"
q	Italic letter (lower case) "cue"
r	Italic letter (lower case) "ahr"
d_{qr}	Italic "dee" subscript "qr"
TS_q	Italic letters (capital) "tee" and "ess" subscript "cue"
CS_q	Italic letters (capital) "cee" and "ess" subscript "cue"
CL_q	Italic letters (capital) "cee" and "ell" subscript "cue"
WL_q	Italic letters (capital) "double u" and "ell" subscript "cue"
$\overline{WL}$	Italic letters (capital) "double u" and "ell" a horizontal bar above
$\vert$	Vertical bar
$\neq$	not equal to
$*$	times
$\%$	percent
O	Italic letter (capital) "oh"
$n^{2.4}$	italic (lower case) "enn" superscript "2.4"
a	Italic letter (lower case) "ay"
$<$	less than
S_0	Italic (capital) "ess" subscript "zero"
C_0	Italic (capital) "cee" subscript "zero"
$\geq$	greater than or equal to
Δ	Greek uppercase delta
T	Italic letter (capital) "tee"
α	Greek lowercase alpha
β	Greek lowercase beta
β^l	Greek lowercase beta superscript "ell"
β^u	Greek lowercase beta superscript "you"

1 Introduction

The task allocation problem is one of assigning the tasks of a parallel program among the processors of a parallel computer in a manner that minimizes inter-processor communication costs while simultaneously maintaining computational load-balance among the processors [2]. In general, given a weighted task interaction graph (V,E) characterizing the parallel program, with vertex-weights representing computational load of the processes and edge-weights capturing the inter-process communication demands, the problem is that of assigning the vertices of the task graph onto processors in a manner that optimizes some cost criterion. This problem is known to be NP-complete except under a few special situations [10, 26]. Hence satisfactory sub-optimal solutions obtainable in a reasonable amount of computation time are generally sought [5,7-11,14,16,18-22]. In this paper, a very efficient algorithm, based on the Kernighan-Lin graph-bisection heuristic [11], is proposed for the task allocation problem in the context of a hypercube parallel computer. The effectiveness of the algorithm is evaluated by comparing the quality of mappings obtained with those derived using simulated annealing [4, 12, 14] on the same sample problems.

The approach proposed in this paper uses a recursive divide-and-conquer strategy. The optimality criterion used is the total weighted inter-processor communication cost under the mapping, subject to the constraint that the computational loads on the processors are balanced to within a specified tolerance. Repeated recursive bipartitioning of the task graph is performed, with the partition at the k^{th} level determining the k^{th} bit of each task's processor assignment.

The effectiveness of the proposed recursive allocation scheme is evaluated by comparing the mappings obtained on test task graphs with those obtained using the well known probabilistic optimization technique of simulated annealing. Since the task allocation problem with arbitrary weights for the tasks (graph vertices) has to be viewed as a constrained optimization problem and the simulated annealing technique cannot be directly applied to constrained optimization, the load-balancing constraint has to be incorporated through use of a *penalty* term in the optimized cost function. The coefficient used for this penalty term is very critical to the quality of the solutions obtained with simulated annealing - too low a value results in violation of the constraints while an excessively high value results in *local optima traps* for finite-time annealing [17]. A *scaled* annealing approach [17] is therefore used to determine an effective value for the penalty-term coefficient. The proposed recursive allocation scheme is shown to produce very good mappings in a significantly shorter time (by several orders of magnitude) than simulated annealing.

The paper is organized as follows. We begin by explaining the mapping problem and the formulation of the cost function in Section 2. Section 3 details the proposed recursive allocation scheme; the scheme is compared with the two phase approach. Section 4 elaborates on the empirical scaled approach taken in applying simulated annealing to the task allocation problem. Section 5 compares the recursive allocation approach to the use of simulated annealing, with respect to quality of produced mappings, as well as the computation time requirements. A brief summary in Section 6 concludes the paper.

2 The Mapping Problem

In this section, we formalize the mapping problem considered and develop the cost function that we attempt to minimize. The parallel program is characterized by a Task Interaction Graph $G(V, E)$, whose vertices, $V = \{1, 2, \ldots, N\}$, represent the tasks of a program, and edges E, correspond to the data communication dependencies between those tasks. The weight of a task i, denoted w_i, represents the computational load of the task. The weight of an edge (i, j) between i and j, denoted c_{ij}, represents the relative amount of communication required between the two tasks.

The parallel computer is represented as a graph $G(P, E_p)$. The vertices $P = \{1, 2, \ldots, K\}$ represent the processors and the edges E_p represent the communication links. The system is assumed to be homogeneous, with all processors being equally powerful and all communication links capable of the same rate of communication. Hence, in contrast to the Task Interaction Graph (TIG), no weights are associated with the vertices or edges of the Processor Interconnection Graph. The processors are assumed to either execute a computation or perform a communication at any given time, but not simultaneously do both. The cost of a communication is assumed to be proportional to the size of the message and the distance between the sender and receiver. The distance d_{qr} between processors q and r is defined as the minimum number of links to be traversed to get from q to r, i.e. it is the length of the shortest path from q to r. By definition, $d_{qr} = 0$ if $q = r$.

The task-to-processor mapping is a function $M : V \rightarrow P$. $M(i)$ gives the processor onto which task i is mapped. The **Task Set** (TS_q) of a processor q is defined as the set of tasks mapped onto it:

$$TS_q = \{j|M(j) = q\}, \quad q = 1, \ldots, K$$

The **Work Load** (WL_q) of processor q is the total computational weight of all tasks mapped onto it:

$$WL_q = \sum_{j \in TS_q} w_j, \quad q = 1, \ldots, K$$

and the idealized average load is given by $\overline{WL} = \frac{1}{K} \sum_{i=1}^{K} WL_i$. The **Communication Set** (CS_q) of processor q is the set of the edges of the Task Interaction Graph that go between it and some other processor under the mapping M:

$$CS_q = \{(i,j)|M(i) = q \quad \text{and} \quad M(j) \neq q\}, \quad q = 1, \ldots, K$$

The **Communication Load** (CL_q) of processor q is the total weighted cost of the edges in its Communication Set, where each edge is weighted by the physical path length to be traversed under the mapping M:

$$CL_q = \sum_{(i,j) \in CS_q} c_{ij} * d_{M(i)M(j)}, \quad q = 1, \ldots, K$$

Cost functions that have been used with the task allocation problem may be broadly categorized as belonging to one of two models: a *minimax cost* model [15, 19, 23] or a *summed total cost* model. With the minimax cost model, the total time required (the execution time + communication time) by each processor under a given mapping is estimated and the maximum cost(time) among all processors is to be minimized:

$$\min_M \left\{ \max_q \left(K_c^{MM} * CL_q + K_e^{MM} * WL_q \right) \right\}, \quad q = 1, \ldots, K \tag{1}$$

where K_c^{MM} and K_e^{MM} are proportionality constants reflecting the relative cost of a unit of communication and a unit of computation (execution) respectively. The cost here does not include synchronization delays.

The summed-total-cost model may be motivated as follows. Ideally, the total computational load should be uniformly distributed among all processors and no communication costs should be incurred at all. Any mapping in practice will of course not match this ideal. The merit of a mapping may be measured in terms of its deviation from the ideal. With respect to load distribution, this may be expressed as the sum among all processors of (the absolute values of) the deviation of the actually assigned load and the known ideal average load. With respect to communication, since in the ideal case we would have no communication at all, the total communication load in the system serves as a good measure of the deviation from the ideal.

$$\text{Cost(M)} = \text{Penalty for communication} + \text{Penalty for computation imbalance}$$

$$\min_{M} \left(K_c^{SC} * \sum_{i=1}^{K} CL_i + K_e^{SC} * \sum_{i=1}^{K} \left| WL_i - \overline{WL} \right| \right) \tag{2}$$

where K_c^{SC} and K_e^{SC} are proportionality constants reflecting the relative penalties for communication and computational load imbalance respectively. Whereas K_c^{MM} and K_e^{MM} used with the minimax cost model capture the physical system parameters of inter-processor communication latency per word and instruction cycle time respectively, a physical interpretation for K_c^{SC} and K_e^{SC} under the summed cost model is not as readily given.

Between these two approaches to modeling the effectiveness of a mapping, the minimax model is the conceptually more accurate one. However, in practice, it is the more difficult one to work with, especially in the context of local-search based optimization techniques, where a cost measure that is incrementally computable in a distributed fashion is attractive. Hence many studies [7, 8, 13, 17, 18, 22] have used some form of a summed cost model in preference to the minimax model. The choice of K_c^{SC} and K_e^{SC} has typically been rather arbitrary. In order to avoid such arbitrary choices for the relative values of K_c^{SC} and K_e^{SC}, in this study, a slightly different summed cost criterion is used – minimization of the summed communication cost alone, subject to load balancing (within a specified tolerance):

$$\min_{M} \left(\sum_{q} CL_q \right), \tag{3}$$

subject to the load-balancing constraint:

$$\frac{\left| WL_q - \overline{WL} \right|}{\overline{WL}} < tol, \qquad q = 1, \ldots, K$$

Thus the desirability of load balancing is decoupled from the communication cost measure. Such an approach is appealing since an acceptable value of *tol*, say 5%, can easily be chosen in practice, whereas a meaningful relative ratio for K_c^{SC}/K_e^{SC} in (2) is not as readily determined.

3 Task Allocation by Recursive Mincut (ARM)

Kernighan and Lin [11] proposed an extremely effective *mincut* heuristic for graph bisection, with an empirically determined time complexity of $O(n^{2.4})$. Their algorithm is based on finding a favorable sequence of vertex-exchanges between the two partitions to minimize the number of inter-partition edges. The evaluation of sequences of perturbations instead of single perturbations endows the method with *hill-climbing* ability, rendering it superior to simple local search heuristics. Fiduccia and Mattheyses [6] used efficient data structures and vertex displacements instead of exchanges to derive a linear time heuristic for graph partitioning, based on a modification of the algorithm in [11]. While the original mincut algorithm of Kernighan and Lin applied only to graphs with uniform vertex weights, the Fiduccia-Mattheyses scheme can handle graphs with variable vertex weights, to divide it into partitions with equi-total vertex weights.

3.1 Two phase approach using mincut heuristic

The mincut bipartitioning procedure can be used recursively to perform a K-way partition of a graph if K is a power of 2 - by first creating two equal sized partitions, then independently dividing each of these into two subpartitions each, and so on till K partitions are created. Such a K-way graph partitioning procedure can be used for performing task allocation on a hypercube using a two-phase approach, referred to as 2PM here on [18]:

1. **Task Clustering:** balanced partitioning of the task graph into K equal sized clusters by recursive bipartitioning, attempting to minimize *inter-cluster* communication volume, and,

2. **Processor Assignment:** assignment of each of the K clusters to one of the K processors attempting to minimize *inter-processor* communication costs.

Such a division of the task-to-processor mapping problem into two subproblems has the advantage that any of the various graph partitioning approaches [6, 11] can be used for the mapping problem, but it also has shortcomings. Due to the decomposition of the problem, even if each subproblem is optimally solvable (which is not the case for general task graphs, since both the graph partitioning problem of step 1 and the graph isomorphism problem involved in step 2 are known to be NP-complete), the mapping problem may not be optimally solved. Fig. 1 provides a specific example to illustrate this point. A simple regular task graph is shown with $2a^2$ nodes, interconnected in an $a \times 2a$ rectangular mesh. The optimal bisection of this graph to minimize the cut separates it into two $a \times a$ meshes. An optimal second-level bisection will split each $a \times a$ mesh into two $a/2 \times a$ meshes. If the second level bisections are independently performed, the configuration shown in Fig. 1(a) can result. In performing the second *processor assignment* step, it is impossible to assign clusters to processors of a 2-dimensional hypercube so that all communication is between directly connected neighbor processors. Thus the minimum summed communication cost that can be achieved is $3.5a$, as shown. If on the other hand, the two second-level bisections performed identical cuts, as shown in Fig. 1(b), then the total communication cost that results is only $3a$, for either choice of optimal cut.

3.2 Direct approach using mincut heuristic

The example illustrates the disadvantage of performing the partitioning and the processor assignment independently in two distinct phases. The task allocation algorithm proposed here merges the two phases, and is hence called as a *direct* approach. The essential idea is to make partial processor assignments to the vertices of the task graph during the recursive bipartitioning steps. At level k in this process, for each vertex, the k^{th} bit of the address of its processor assignment is determined. Equivalently, the bisections at level k may be viewed as successively refining the subcube to which a vertex is to be assigned. Initially, prior to any partitioning, the entire hypercube is the single subcube under consideration and each vertex clearly is to be assigned within this subcube. The first bipartitioning of the task graph separates the vertices into two groups, each to be assigned to a distinct subcube of size $K/2$, i.e. the highest order bit of the processor to be assigned to, is uniquely determined. At each succeeding level, during bipartitioning, edge costs are weighted by the number of differing bits in the partial processor assignments of the two relevant vertices. For the example shown, the first bisection at level 2 will be arbitrarily made, as with the 2-phase approach. However, when the second level-2 bisection is made, when considering node transfers, the costs due to edges going across to vertices earlier assigned at this level will be weighted appropriately. Consequently, one of the two configurations shown in Fig. 1(b) results rather than a configuration as in Fig. 1(a).

The task allocation algorithm, termed *Allocation by Recursive Mincut* (ARM), is sketched in

C-pseudocode in Fig. 3. Repeated, recursive bipartitioning is performed, using procedure *mincut*. After each bipartitioning step, one bit is set in the processor assignment of the vertices involved in the partitioning step, according to the outcome of the partition. The mincut procedure comprises three phases: the first phase performs the basic cut minimization, using the procedure *calc_GAIN* (shown in Fig. 2) to compute the improvement in weighted communication cost due to a trial node transfer; the second phase performs fine-tuning, using the same mincut heuristic, but now allowing node transfers from either partition, using a composite measure that incorporates load-imbalance as well as communication costs – this is done in order to improve load-balance in case the resulting partition at the end of phase 1 was not very well balanced; finally, if the partition resulting at the end of phase 2 is not load-balanced to within the required tolerance, phase 3 attempts to achieve balance, at the price of added communication cost. The different phases of mincut algorithms are described below:

Cut-Minimization Phase(1)

The basic mincut algorithm used here is similar in spirit to the Fiduccia-Mattheyses variant of the Kernighan-Lin heuristic. An initial two-way partition is created by assigning the nodes of the graph, one by one (in decreasing order of node weights) always to the partition with smaller total weight (randomly in case both are equal). This results in a load-balanced initial partition. Phase 1 of the algorithm executes several iterations until there is no improvement as a result of an iteration. At the beginning of each iteration, all nodes are marked as locked and the gain value of each node is calculated. The gain value of a node is the resulting reduction in the cost of the cut due to the transfer of the node from the current partition to the other partition. After creating the initial load-balanced partition, a sequence of maximally improving node transfers from the partition with currently greater load to the partition with lower load are tried. On each transfer, a node with the highest gain i.e. the one which reduces the sum of the weights of the edges cut maximally, is selected. After each transfer, the selected vertex is locked to prevent it from being chosen more than once in a sequence. Assuming that the chosen node is transferred, the gains of all unlocked nodes are updated and a sequence number is associated with the transfer. The cumulative gain of a sequence of node transfers is evaluated for each sequence number and the the sequence number that maximizes the cumulative gain is determined. If the maximum cumulative gain is greater than zero or if the maximum cumulative gain is zero but the node transfers lead to better load-balancing, all node transfers with index between 1 and the maximizing sequence number are performed and a new iteration is begun.

Fine-tuning Phase(2)

The second phase of the algorithm attempts to improve load-balancing. The computation in this phase is similar to that in phase 1 except that the gain of each node includes a component indicating the improvement in load-balance as a result of the node transfer in addition to the reduction in the cost of the cut. Thus, for example, among nodes that reduce the cost of the cut to the same extent, node transfers from the heavier partition to the lighter partition are favored.

Rebalancing for Tolerance Phase(3)

Phase 3 is invoked only if the sum of weights of vertices in one partition exceeds that of the other beyond the tolerance level. A minimal sequence of vertex transfers that brings the load imbalance within tolerance is determined and the transfers are performed as follows: for each node in the heavier partition, the improvement in load balance is computed and the node with the greatest improvement is chosen and transferred to the other partition. If the load imbalance is within tolerance, the procedure terminates, otherwise, it is repeated using the nodes of the heavier partition.

The time complexity of ARM is $O(|V| * \log_2 K)$, because the depth of recursion is $\log_2 K$ and each level has a total of $|V|$ vertices to work on.

3.3 Comparison of 2-phase (2PM) and Direct (ARM) approaches

In this section, we compare 2PM and ARM in terms of solution quality. Seven test task graphs were used, five of them representative of task graphs arising from finite element applications [7, 19], and two randomly generated graphs. The target structure is an 8-node hypercube. The experiments were performed on a Pyramid 9825 processor running the Pyramid OSx/4.0 operating system. For each method, ten runs were performed, starting each time with a random initial configuration.

Table 1 presents a summary of the results obtained, with respect to the quality of mappings generated; it presents the minimum, mean and maximum costs obtained for the sample runs as well the standard deviation of the costs. The mappings generated by ARM were generally slightly better ($< 5\%$) for the first five sample graphs from finite element applications, whereas the solutions from

ARM for the last two samples were significantly better (around 10%). For larger hypercubes, the difference between 2PM and ARM is significant for all samples.

4 Simulated Annealing (SA)

Simulated annealing is a powerful general purpose combinatorial optimization technique proposed in [4, 12, 14]; this is an extension of a Monte Carlo method developed by Metropolis et al. [16] to determine the equilibrium state of a collection of particles at any given temperature. The approach is based on an analogy between the annealing process in which a material is melted and cooled very slowly and the solution of difficult combinatorial optimization problems. Its basic feature is the ability to explore the configuration space of the problem allowing controlled hill-climbing moves (changes to a configuration that worsen the solution) in an attempt to reduce the probability of getting stuck at high-lying local minima. The acceptance of hill-climbing moves is controlled by a parameter, analogous to the temperature of the material in the annealing process, that makes them less and less likely toward the end of the process. In abstracting the method to solve combinatorial optimization problems, the objective function or the cost to be optimized is identified with the energy in the annealing process. The method starts with a random initial configuration, S_0, which has a certain cost associated with it, C_0. A new configuration, S, is generated by a perturbation of S_0, resulting in a new cost, C. The change in cost $\Delta C = C - C_0$, is estimated or calculated; if $\Delta C < 0$, the move is accepted; if $\Delta C \geq 0$, then the move is accepted with a probability $\exp(-\Delta C/T)$, where T is the parameter that controls the hill-climbing; the parameter T, called temperature, is gradually reduced during the execution of the algorithm.

The annealing algorithm is characterized by the following:

- the perturb function,
- the acceptance criterion which has been explicitly stated here,
- the temperature update function which is typically of the form $T_{new} = \alpha(T) * T$, where α is a function of temperature and $0 < \alpha(T) < 1$,
- the equilibrium condition at current temperature, which is usually referred to as the inner-loop criterion, and
- the freezing point condition, usually referred to as the stopping criterion.

The inner-loop criterion is in general specified as a certain number of iterations of the inner loop i.e. the number of attempted new configurations at a given temperature; this number should be high enough to allow the system to come to equilibrium at the current temperature. The stopping criterion is usually a certain "low" temperature (T_{stop}) near the "freezing" point of the system i.e. a small positive value of temperature where the acceptance probability of hill-climbing moves is extremely low. It is interesting to observe that if T were set equal to infinity, the above algorithm is nothing but a totally randomized algorithm for searching through the configuration space; and if T were set equal to zero, no hill climbing moves are accepted, giving rise to the iterative improvement or local search algorithm. It has been observed that a high constant value of α (around 0.95) has yielded consistently good results for many applications of simulated annealing [21, 22].

The implementation of simulated annealing described here uses starting or initial configurations that are generated by random allocation of tasks among processors. The starting temperature T_0 (or T_{start}) is then determined so as to give an acceptance probability of 0.9 for the mean increase in the cost function, for all possible changes to the initial configuration resulting from a single move from that configuration i.e. moving a task from one processor to any processor. With V vertices in the graph and K processors available for the assignment, a single vertex could be moved from one processor to any of the remaining $K-1$ processors, giving rise to a total of $N = V*(K-1)$ possible neighboring moves. The freezing point is set so that a move increasing the cost function by a unit value has an acceptance probability of 2^{-31}. The inner-loop criterion i.e. the number of moves attempted at each temperature to allow the system to attain equilibrium at that temperature is specified as a multiple (M) of N; it is set to $M*N$. It has been observed experimentally that as the value of M is increased from a small value such as 0.01, the quality of solutions obtained improves till a certain value of M, say 1, after which further increase of M results in no significant improvement in the solution quality; in addition, the running time of simulated annealing is proportional to the value of M. The sequence of values chosen for temperature T through the update function is known as the cooling schedule. The cooling schedule used here is given by

$$T_{new} = 0.95 * T,$$

i.e. after attempting $M*N$ moves at T the temperature is lowered to T_{new}. The time complexity of simulated annealing for task allocation as implemented is $O(M*|V|*K*\log(T_{start}/T_{stop}))$.

4.1 Constrained Optimization by Simulated Annealing

The simulated annealing algorithm outlined in the previous section is applicable to the *unconstrained* optimization of an objective function. However, the mapping problem addressed here requires the minimization of inter-processor communication costs, subject to the load-balance constraint. The way to incorporate such constraints in the application of simulated annealing is through the addition of *penalty terms* in the function being minimized, so that violation of any constraint(s) results in a significant contribution to the total cost function from the penalty term(s). For the task allocation problem, besides the communication cost to be minimized, an additional penalty term proportional to the sum of load-deviations from the ideal average is added, to give a two-part cost function similar to equation 2 in Section 2:

$$Cost(M) = \sum_{i=1}^{K} CL_i + \beta * \sum_{i=1}^{K} \left| WL_i - \overline{WL} \right| \tag{4}$$

The value of the coefficient β used in such a two-part summed cost function is crucial in determining the quality of generated solutions. Clearly, if β is very small, the penalty term will make an insignificant contribution to the total cost and consequently be ineffective in generating mappings that satisfy the load-balance constraint. On the other hand, if β is very large, the penalty term will have a dominant effect and the resulting mappings may have relatively high communication costs. In fact, a large β has a detrimental effect even when mapping task graphs that have perfectly load-balanced optimal mappings. This is a consequence of a *local optimum trap* phenomenon that manifests itself with finite-step simulated annealing. This phenomenon is explored in greater detail in [17] and is briefly explained below.

An examination of the move acceptance/rejection trace of annealing runs with a high β reveals that the annealing algorithm invariably gets “trapped” in a configuration that is almost perfectly load-balanced, but has high total communication costs. Whereas a sequence of moves from this configuration could lead to a cumulative cost improvement, any single move would only result in an overall increase in cost due to the increased load-imbalance cost in going from a balanced configuration to a load-imbalanced configuration. Except at high temperatures, the probability of acceptance of any move out of such a local optimum trap is extremely small. The annealer thus gets trapped into a local optimum configuration and stays in that configuration for a significant fraction of the tail-end of the cooling schedule, resulting in poor solutions. As might be expected, this tendency to fall into a local optimum trap is dependent on the value of β used - the higher its value, the greater the tendency to get stuck at a local optimum.

Two approaches to avoiding such local optima traps are evaluated in [17]. One approach involves the empirical determination of the relative values for the coefficients of the terms of the two-part cost-function that results in the best mappings. This can be done by performing simulated annealing runs for chosen values of β. As β is decreased, the communication cost of the obtained mapping tends to decrease. As β becomes very low, however, the load-balance constraint gets violated. The following strategy is used to select β values. Starting with some initial value of β, say 1, simulated annealing is tried and β doubled if the obtained solution violates the load-balance criterion. This value of β, β^u, is an upper bound on the optimal value of β to be used. Starting with the previous tried value of β (or 0 if the first trial provided a valid mapping) as β^l, the interval between β^l and β^u is repeatedly halved, with one of the two end-points being changed to the newly tried mean value, until a termination criterion (e.g. three successive invalid mappings) is met. The final value of β^u is taken as the optimal value to use. Since a number of trial runs are required, the computation time required is increased. However, the trial runs can be performed with a much smaller value of M (say one tenth) than the final optimization runs.

An alternate approach is to use vertex swaps as the configuration perturbation mechanism for simulated annealing instead of vertex displacements. If this is done, for the case of graphs with uniform vertex weights, it is easy to see that the penalty term in (4) will stay unchanged for all configurations reachable from the initial configuration - thus the use of a two-part cost-function will not lead to local optima traps during simulated annealing. Even when the vertex weights of the task graph are variable, if a perfectly load-balanced configuration is reached, there will likely be many other load-balanced configurations that can be reached from the current one by a single task-exchange. It is found in practice [17] that local optima traps do not occur with this approach in general, but the quality of solutions obtained with the scaled approach is consistently superior. Hence the scaled annealing approach is used in deriving mappings for comparison with solutions obtained with the proposed recursive allocation scheme.

In the case of the scaled simulated annealing approach, different values of M were tried. Recall from section 4 that the value of M prescribes the inner-loop criterion in the implementation described here *i.e.* the number of moves attempted at each temperature is $M * N$, where $N = V(K-1)$ and V is the number of vertices in the task graph and K is the number of processors. The results are given in Table. 2 – the table presents results for three different values of M, namely 1, 5 and 15. At $M = 1$, SA produced very poor quality mappings but as the value of M is increased, the solution quality is significantly better, as expected. At $M = 15$, we observe the best solutions we have obtained for several sample graphs.

5 Comparison of ARM and SA

In this section, we compare ARM and SA in terms of solution quality and running times. Seven test task graphs were used, five of them representative of task graphs arising from finite element applications [7, 19], and two randomly generated graphs. The target structure is an 8-node hypercube. The experiments were performed on a Pyramid 9825 processor running the Pyramid OSx/4.0 operating system. For each method, ten runs were performed, starting each time with a random initial configuration. We note that each method produces mappings that are significantly better than random mappings. The mappings generated by SA were generally slightly better, although the solutions obtained by ARM were never worse by more than 10% in terms of total communication costs. We note from section 4.1 that the quality of solutions from SA is a function of M.

Table 3 presents the running times required for each test case by the following methods: 2PM, ARM and SA for $M = 1$, $M = 5$ and $M = 15$. It can be seen that 2PM and ARM are over one hundred times as fast as SA for $M = 5$. A smaller value of M results in a proportionately smaller running time. However, with M=1, the solutions obtained are poorer than ARM's mappings, while the running times are at least twenty times worse. The running times required by SA are thus excessive. In fact, in the case of the samples motivated from finite element analysis, the time taken to perform the mapping was larger than the time that would be required for a typical finite element run on such a sample! ARM is thus clearly preferable in a practical context.

6 Conclusions

A computationally efficient approach to mapping task graphs onto a hypercube parallel computer was presented. The algorithm was based on a recursive divide-and conquer strategy, using a variant of the Kernighan-Lin min-cut bi-partitioning heuristic. The effectiveness of the scheme was evaluated by comparison with the combinatorial optimization technique of simulated annealing. The constrained optimization problem of balanced mapping of a task graph onto a hypercube was modeled for simulated annealing using a penalty term in the optimization function to enforce the load-balance constraint. A problem with severe local optima traps to finite-step annealing led to the use of an adaptive, scaled annealing approach. The quality of solutions produced by the recursive allocation approach was within 10% of the solutions from simulated annealing, but required less than one hundredth the computation time.

References

[1] Berman, F. "Experience with an Automatic Solution to the Mapping Problem." In Jamieson, L. H., Gannon, D. B., and Douglass, R. J.(Eds). *The Characteristics of Parallel Algorithms.* MIT Press, Cambridge, MA, 1987, pp. 307-334.

[2] Bokhari, S. H. "On the Mapping Problem." *IEEE Trans. on Computers.* Vol. C-30, No. 3, Mar. 1981, pp. 207-214.

[3] Bollinger, S. W., and Midkiff, S. F. "Processor and Link Assignment in Multicomputers using Simulated Annealing." *Proc. 1988 International Conference on Parallel Processing.* Vol. I Architecture, pp. 1-7.

[4] Cerny, V. "Minimization of Continuous Functions by Simulated Annealing." Research Report, Research Institute for Theoretical Physics, University of Helsinki, No. HU-TFT-84-51, 1984.

[5] Efe, K. "Heuristic Models of Task Assignment Scheduling in Distributed Systems." *Computer.* Vol. 15, No. 6, Jun. 1982, pp. 50-56.

[6] Fiduccia, C. M., and Mattheyses, R. M. "A Linear-Time Heuristic for Improving Network Partitions." *Proc. of the 19th Design Automation Conference.* Jun. 1982, pp. 175-181.

[7] Flower, J. W., Otto, S. W., and Salama, M. C. "A Preprocessor for Irregular Finite Element Problems." Tech. Rep. Caltech Concurrent Computation Project. Report #292, Jun. 1985.

[8] Fox, G. C. "Load Balancing and Sparse Matrix Vector Multiplication on the Hypercube." Tech. Rep. Caltech Concurrent Computation Project. Report #327, Jul. 1985.

[9] Fox, G. C., and Otto, S. W. "Concurrent Computation and the Theory of Complex Systems." In Moler, C. B. (Ed) *Hypercube Multiprocessors 1986.* SIAM, Philadelphia, PA, 1987, pp. 244-268.

[10] Kasahara, H., and Narita, S. "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing." *IEEE Trans. Computers.* Vol. C-33, No. 11, Nov. 1984, pp. 1023-1029.

[11] Kernighan, B. W., and Lin, S. "An Efficient Heuristic Procedure for Partitioning Graphs." *Bell Systems Technical Journal.* Vol. 49, No. 2, 1970, pp. 291-308.

[12] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. "Optimization by Simulated Annealing." *Science.* 220 (1983) 671-680.

[13] Kramer, O., and Muhlenbein, H. "Mapping Strategies in Message-based Multiprocessor Systems." *Proc. PARLE 87.* Vol. 1, Lecture Notes in Computer Science No. 258, Springer-Verlag, Berlin, Jun. 1987.

[14] van Laarhoven, P. J. M., and Aarts, E. H. L. *Simulated Annealing: Theory and Applications.* D. Reidel Publishing Company, 1987.

[15] Lo, V. M. "Heuristic Algorithms for Task Assignment in Distributed Systems." *IEEE Trans. Computers.* Vol. C-37, No. 11, Nov. 1988, pp. 1384-1397.

[16] Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., and Teller, E. "Equation of State Calculations by Fast Computing Machines." *J. Chemical Physics.* **21** (1953) 1087-1092.

[17] Ramanujam, J., Ercal, F., and Sadayappan, P. "Task Allocation by Simulated Annealing." *Proc. International Conference on Supercomputing.* Boston MA, May 1988, Vol. III Hardware & Software, pp. 471-480.

[18] Sadayappan, P., and Ercal, F. "Cluster Partitioning Approaches to Mapping Parallel Programs onto a Hypercube." *Proc. International Conference on Supercomputing.* Lecture Notes in Computer Science No. 297, Springer-Verlag, Berlin, Jun. 1987, pp. 475-497.

[19] Sadayappan, P., and Ercal, F. "Nearest-Neighbor Mapping of Finite Element Graphs Onto Processor Meshes." *IEEE Trans. on Computers.* Vol. C-36, No. 12, Dec. 1987, pp. 1408-1424.

[20] Schwan, K., and Gaimon, C. "Automating Resource Allocation in the Cm^* Multiprocessor." *Proc. 5th International Conference on Distributed Computing Systems.* May 1985, pp. 310-320.

[21] Sechen, C., and Sangiovanni-Vincentelli, A. "The TimberWolf Placement and Routing Package." *IEEE J. Solid-State Circuits.* Vol. SC-20, No. 2, Apr. 1985, pp. 510-522.

[22] Shield, J. "Partitioning Concurrent VLSI Simulation Programs onto a Multiprocessor by Simulated Annealing." *IEE Proceedings.* Part G, Vol.134, No.1, Jan. 1987, pp. 24-28.

[23] Shen, C., and Tsai, W. "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion." *IEEE Trans. on Computers.* Vol. C-34, No. 3, Mar. 1985, pp. 197-203.

[24] Sinclair, J. B., and Lu, M. "Module Assignments in Distributed Systems." *Proc. 1984 Computer Networking Symposium,* Gaithersburg, MD, Dec. 1984, pp. 105-111.

[25] Sinclair, J. B. "Efficient Computation of Optimal Assignments for Distributed Tasks." *J. of Parallel and Distributed Computing.* Vol. 4, No. 4, Aug. 1987, pp. 342-362.

[26] Stone, H. S. "Multiprocessor Scheduling with the Aid of Network Flow Algorithms." *IEEE Trans. Software Engineering.* Vol. SE-3, No. 1, Jan. 1977, pp. 85-93.

Figure 1: 2-phase vs. Direct Approach to Task Allocation.

Algorithm ARM $(V, tol, depth, Maxdepth = \log_2 C, S_C)$

```
/* Allocation by Recursive Mincut */
/* V : vertex set of the graph G = (V, E) to be partitioned into C clusters */
/* S_C : Set of clusters obtained */

if (depth = Maxdepth) then
    /* no more bisection of V ; add it to the set of clusters S_C */
    S_C ← S_C ∪ V
else /* recursively partition */
{
    (origC_1, origC_2) ← initpart(V); /* divide V into two equal-weight initial partitions */
    (C_1, C_2,) ← mincut(origC_1, origC_2, tol, depth);
    - Set the k^th bit of map[v] for all v ∈ C_1 to 0;
    - Set the k^th bit of map[v] for all v ∈ C_2 to 1;
    ARM(C_1, tol, depth + 1, Maxdepth, S_C);
    ARM(C_2, tol, depth + 1, Maxdepth, S_C);
}
```

Figure 2: Recursive bisection algorithm

Algorithm calc_GAIN(v, C_1, C_2, k)

```
/* A global adjacency matrix for the entire graph is used. This routine has access */
/* to all neighbors of v and information about them to calculate the exact gain for */
/* vertex v during any stage of the recursive bipartitioning process */
/* It is assumed that k^th bit of map[v] is 0, ∀v ∈ C1 and k^th bit of map[v] is 1, ∀v ∈ C2. */

g_v ← 0 /* initialize the gain of the vertex v to zero */
for (each neighbor v_i of v) do
    if (v_i is in the same cluster where v is) then
        g_v ← g_v − cost(v, v_i) /* subtract edge-weight from gain */
    else
        if (edge (v_i, v) is in the cut between C1 and C2) then
            g_v ← g_v + cost(v, v_i) /* add edge-weight to the gain */
        else /* v_i is neither in C1 nor in C2 */
            if (the k^th bit in map[v] is already set) then
                /* gain is based on the hamming distance between partial */
                /* mappings of v and v_i considering only the k^th bit : */
                if (the k^th bit of map[v] and map[v_i] are identical) then
                    /* distance will increase by 1 when v is moved to the opposite cluster */
                    g_v ← g_v − cost(v, v_i) /* subtract edge-weight from gain */
                else /* the k^th bit of map[v] and map[v_i] are different */
                /* distance will decrease by 1 when v is moved to the opposite cluster */
                    g_v ← g_v + cost(v, v_i) /* add edge-weight to the gain */
                endif
            endif
        endif
    endif
endfor /* each neighbor */
```

Figure 3: The calc_GAIN algorithm

$T \leftarrow T_0$;
$S \leftarrow S_0$;
$C \leftarrow C_0$;
while ("the freezing point has not yet been reached") **do** {
    while ("equilibrium at current temperature has not yet been reached") **do** {
        $S_{new} \leftarrow perturb(S)$;
        $C_{new} \leftarrow$ estimate of the cost the new configuration, S_{new};
        $\Delta C \leftarrow C_{new} - C$;
        if $\Delta C < 0$ **then** {
            $S \leftarrow S_{new}$;
            $C \leftarrow C_{new}$
        } **else** {
            $r \leftarrow$ random number between 0 and 1;
            if $r < \exp -\Delta C/T$ **then** {
                $S \leftarrow S_{new}$;
                $C \leftarrow C_{new}$;
            }
        }
    }
    $T_{new} \leftarrow update(T)$
}

Figure 4: Simulated Annealing Algorithm

Comparison of solution quality of ARM & 2-phase mincut										
No.	Graph	Min Cost		Mean Cost		Max Cost		Std Dev		
	$\|V\|$	2PM	ARM	2PM	ARM	2PM	ARM	2PM	ARM	
1.	144	102	100	106.1	106.1	112	112	3.3	4.1	
2.	192	80	80	99.9	91.3	125	117	16.9	11.4	
3.	256	69	64	77.5	69.2	94	80	7.8	5.7	
4.	505	166	163	175.9	175.9	211	203	12.8	12.6	
5.	602	254	252	278.0	280.0	312	305	17.5	15.2	
6.	200	16	16	16.6	16	19	16	1.2	0.0	
7.	400	221	205	233.6	216.4	246	223	8.4	4.6	

Table 1: Comparison of solution quality of ARM and 2-phase mincut on sample graphs

Comparison of solution quality of SA for different values of M													
No.	Graph	Min Cost			Mean Cost			Max Cost			Std. Dev.		
	$\|V\|$	1	5	15	1	5	15	1	5	15	1	5	15
1.	144	98	98	98	102.9	99.7	98.0	110	105	98	5.1	2.7	0.0
2.	192	81	80	80	104.1	94.7	90.0	124	120	100	14.1	12.3	10.0
3.	256	66	75	64	84.0	80.6	69.5	94	85	78	7.8	2.8	5.6
4.	505	169	158	157	234.5	173.7	169.0	277	202	180	35.6	17.9	10.8
5.	602	264	252	241	343.4	282.3	259.2	402	347	288	33.3	26.6	15.3
6.	200	20	18	19	23.6	23.9	21.6	28	29	26	2.5	3.5	2.1
7.	400	196	185	178	212.6	192.8	185.6	226	198	200	9.0	3.5	5.9

Table 2: Comparison of solution quality of SA for different values of M

Comparison of running times of ARM & SA							
	Graph Characteristics		Running Times in sec.				
No.	$\|V\|$	Description	Mincut		Simulated Annealing		
			2PM	ARM	M=1	M=5	M=15
1.	144	U-shaped mesh	4.9	4.9	371.7	1743.5	5169.4
2.	192	Donut 8-point	6.6	6.6	485.7	2319.0	7055.9
3.	256	Regular mesh	7.4	7.4	475.4	2384.6	6841.4
4.	505	Irregular mesh	22.9	24.7	1388.2	6486.1	19391.8
5.	602	Plate non-mesh	29.3	30.1	1673.3	7928.5	23652.6
6.	200	Random	4.9	5.0	386.0	1850.3	5390.9
7.	400	Random	11.8	11.8	733.6	3487.4	10843.2

Table 3: Comparison of running times of ARM and SA on sample graphs

Biography of Fikret Ercal:

Fikret Ercal was born in Konya, Turkey. He received the B.S. (with highest honors) and M.S. degrees in electronics and communication engineering from the Technical University of Istanbul, Turkey, in 1979 and 1981, respectively, and Ph.D. degree in computer and information science from the Ohio State University in 1988.

From 1979 to 1982, he served as a Teaching and Research Assistant in the Department of Electrical Engineering, Technical University of Istanbul. He has been a scholar of the Turkish Scientific and Technical Research Council since 1971. Currently, he is an assistant professor in Bilkent University, Ankara, Turkey. His research interests include parallel computer architectures, algorithms, parallel and distributed computing systems. Dr. Ercal is a member of Phi Kappa Phi.

Biography of J. Ramanujam:

J. Ramanujam received a B. Tech degree in Electrical Engineering from the Indian Institute of Technology, Madras in 1983 and an M. S. degree in computer science from the Ohio State University in 1987, where he currently working towards a Ph. D. degree. His research interests include parallel computer architecture, parallelizing compilers and parallel algorithms.

Biography of P. Sadayappan:

P. Sadayappan received a B. Tech degree from the Indian Institute of Technology, Madras, and an M. S. and Ph. D. from the State University of New York at Stony Brook, all in Electrical Engineering.

Since 1983 he has been an Assistant Professor with the Department of Computer and Information Science, The Ohio State University, Columbus. His research interests include parallel computer architecture, parallel algorithms and applied parallel computing.