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

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.

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 interprocessor communication costs while simultaneously maintaining computational load-balance among the processors. 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 interprocess 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 [11,27]. 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 [12], 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 [4,13,15] 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 [19]. A *scaled* annealing ap-

proach [19] 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. 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:

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

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 [16,21,25,26] 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:

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

where K_c^{MM} and K_e^{MM} are proportionality constants reflecting the relative cost of a unit of commu-

nication and a unit of computation (execution) respectively.

The summed-total-cost approach 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.

$$\begin{aligned}\text{Cost(M)} = \quad & \text{Penalty for communication} + \\ & \text{Penalty for computation imbalance}\end{aligned}$$

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

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 [7,8,10,14,19,20,24] 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):

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

subject to the load-balancing constraint:

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

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 [12] 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 [12]. 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.

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, always to the partition with lesser total weight (randomly in case both are equal). This results in a load-balanced initial 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. The iterative improvement heuristic is otherwise very similar to the Kernighan-Lin mincut heuristic, except for the use of one-way node transfers instead of node exchanges. The use of node transfers

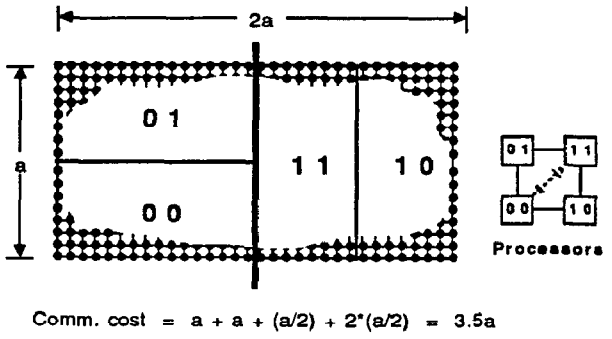
in this fashion guarantees load-balance even though the individual vertex weights are variable.

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 [20]:

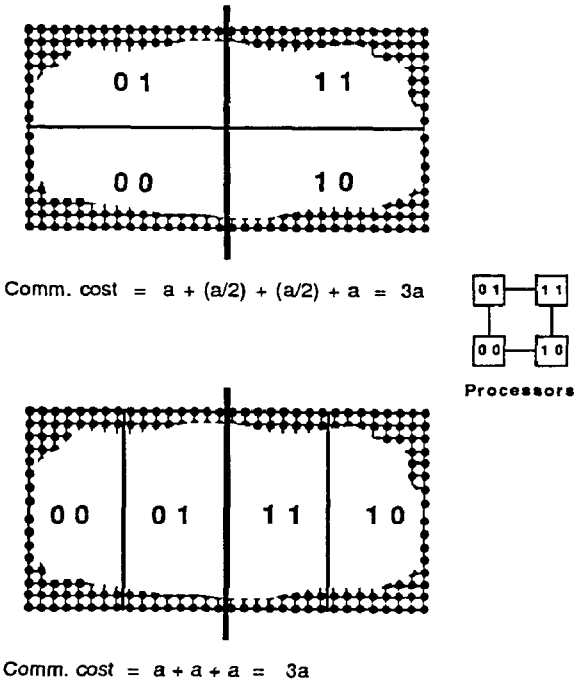
1. **Task Clustering:** balanced partitioning of the task graph into K equal sized clusters by recursive bipartitioning, attempting to minimize *inter-cluster* communication volume, and,
2. **Processor Assignment:** assignment of each of the K clusters to one of the K processors attempting to minimize *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 [1,6,10,12,18] 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.

The example illustrates the disadvantage of performing the partitioning and the processor assignment independently in two distinct phases. The task



(a) 2-phase approach



(b) Direct approach

Figure 1: Illustration of 2-phase vs. direct approach to task allocation.

Algorithm mincut ($orig - C_1, orig - C_2, tol, k = depth$)

/* accepts two clusters $orig - C_1$, $orig - C_2$ as input and tries to reduce the */
/* cutsize between these clusters by moving vertices between them. The algorithm */
/* returns the final clusters. map[v] represents the processor assignment for vertex v*/

- Initially $C_1 \leftarrow orig - C_1$, $C_2 \leftarrow orig - C_2$
- Associate a gain value g_v with each node v and initialize g_v to zero

/* **Phase1:** */
do
{ - Mark all nodes unlocked
- Compute $\forall v \in V, g_v = \text{calc\_GAIN}(v, C_1, C_2, k)$
- Compute W_1 and W_2, the total load in C_1 and C_2 respectively
seqno $\leftarrow$ 0 ; done $\leftarrow$ FALSE
repeat
{
seqno $\leftarrow$ seqno + 1
- Let C_i be the cluster with greater total weight and C_j the lesser total weight i.e., $W_i \geq W_j$, $i, j \in \{1, 2\}$, $i \neq j$
- Among the unlocked vertices, identify $v^* \in C_i$ with maximal gain g_v^*
- If no such vertex exists, done $\leftarrow$ TRUE
- Assume that v^* is moved to C_j, update the gain for all unlocked nodes and calculate loads W_1 and W_2 for C_1 and C_2
- Lock v^* and record the status of the movement and $gain[seqno] \leftarrow g_v^*$
} **until** done
Let $\mathcal{G}^* = \max_l \sum_{i=1}^{l} gain[i] = \sum_{i=1}^{l^*} gain[i]$
i.e., l^* is the value of l that maximizes the cumulative gain
if [($\mathcal{G}^* > 0$) OR (($\mathcal{G}^* = 0$) AND (better load balancing))]
then perform all moves from 1 to l^*
} **while** $\mathcal{G}^* > 0$
return (C_1, C_2,)

/* **Phase2:** */
do
{
Computation virtually identical to that in Phase 1 except that:
The gain associated with a vertex is the sum of the reduction in the
cutsize (as before, obtained by using calc_GAIN())and the improvement
in load balancing; the improvement in load balancing is the reduction
in the sum of the absolute deviations of the load in each partition
from the ideal average
} **while** $\mathcal{G}^* > 0$

/* **Phase3:** */
if (*tol* is not satisfied) **then**
considering the vertex which balances the system most first, move as
many vertices as needed from one partition to the other to balance the
partitions to within the given tolerance, *tol*
endif

Figure 2: Mincut 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

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 */
{
   (origC1, origC2) ← randpart(V); /* randomly divide V into two equal-weight partitions */
   (C1, C2,) ← mincut(origC1, origC2, tol, depth);
   - Set the k^th bit of map[v] for all v ∈ C1 to 0;
   - Set the k^th bit of map[v] for all v ∈ C2 to 1;
   ARM(C1, tol, depth + 1, Maxdepth, S_C);
   ARM(C2, tol, depth + 1, Maxdepth, S_C);
}
```

Figure 4: Recursive bisection algorithm

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. 2–4. 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* 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 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.

4 Simulated Annealing (SA)

Simulated annealing is a powerful general purpose combinatorial optimization technique proposed in [4,13,15]; this is an extension of a Monte Carlo method developed by Metropolis et al., [17] 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 structure of the annealing algorithm is shown in Fig. 5.

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.

```
T ← T0;
S ← S0;
C ← C0;
while ("the freezing point has not yet been reached") do {
   while ("equilibrium at current temperature has not yet been reached") do {
      Snew ← perturb(S);
      Cnew ← estimate of the cost the new configuration, Snew;
      ΔC ← Cnew − C;
      if ΔC < 0 then {
         S ← Snew;
         C ← Cnew
      } else {
         r ← random number between 0 and 1;
         if r < exp −ΔC/T then {
            S ← Snew;
            C ← Cnew;
         }
      }
   }
   Tnew ← update(T)
}
```

Figure 5: Simulated Annealing Algorithm

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 none other than a totally randomized algorithm for searching through the configuration space; and if T were set equal to zero, no hill climing moves are accepted, giving rise to the iterative improvement or local search algorithm. It has been observed that a high constant value of a around 0.95 has yielded consistently good results for many applications of simulated annealing [23,24].

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 improved till a certain value of M, say 1, after which further increase of M resulted 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 $O(M * |V| * K * \log(T_{start}/T_{stop}))$.

Constrained Optimization by Simulated Annealing

The simulated annealing algorithm outlined in the previous section was 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:

$$Cost(M) = \sum_{i=1}^{K} CL_i + K_e * \sum_{i=1}^{K} |WL_i - \overline{WL}| \quad (3)$$

The value of the coefficient K_e used in such a two-part summed cost function is crucial in determining the quality of generated solutions. Clearly, if K_e 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 K_e 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 K_e 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 [19] and is briefly explained below.

An examination of the move acceptance/rejection trace of annealing runs with a high K_e 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 K_e 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 [19]. 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 K_e. As K_e is decreased, the communication cost of the obtained mapping tends to decrease. As K_e becomes very low, however, the load-balance constraint gets violated. The following strategy is used to select K_e values. Starting with some initial value of K_e, say 1, simulated annealing is tried and K_e doubled if the obtained solution violates the load-balance criterion. This value of K_e, K_e^u, is an upper bound on the optimal value of K_e to be used. Starting with the previous tried value of K_e (or 0 if the first trial provided a valid mapping) as K_e^l, the interval between K_e^l and K_e^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 K_e^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 (3) 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 per-

fectly 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 [19] 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.

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 [7,21], and two randomly generated graphs. For each method, ten runs were performed, starting each time with a random initial configuration. The cost of the best obtained mapping and the mean cost of the mappings are both reported for each test case. For each case, the cost of a randomly generated mapping is also reported, to provide a basis for comparing the mappings obtained by SA and ARM.

Table 1 presents a summary of the results obtained, with respect to the quality of mappings generated. Both methods generated mappings that were significantly superior to a randomly generated mapping. 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, and were actually better in a couple of cases. The quality of solution produced by SA is a function of the parameter M. The reported results correspond to a relatively high value of 5 for M. SA was also tried at smaller values of M, and produced poorer mappings. For example, with M=1, the solutions were consistently poorer than the mappings produced by ARM.

Table 2 presents the running times required for each test case by the two methods. It can be seen that ARM is over a hundred times as fast as SA. The running times for SA correspond to M=5. A smaller value of M results in a proportionately smaller running time. However, as observed earlier, 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 using 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. Work is currently in progress in comparing other approaches to the task allocation (graph partitioning) problem such as genetic evolution-based methods [18], neural net-based load balancing techniques [10] and graph partitioning through matrix approximation [1].

Acknowledgements

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).

References

[1] E.R. Barnes, "An Algorithm for Partitioning the Nodes of a Graph," *SIAM J. of Alg. and Dis. Meth,* Vol. 3, No. 4, Dec. 1982, pp.541-550.

[2] M.J. Berger and S.H. Bokhari, "A Partitioning Strategy for Nonuniform Problems on Multipro-

Comparison of solution quality of ARM & SA							
No.	Graph Characteristics		Initial Cost	Min Cost		Mean Cost	
	$\|V\|$	Description	Random	ARM	SA	ARM	SA
1.	144	U-shaped mesh	723	100	98	108.1	102
2.	192	Donut 8-point	937	80	80	95.3	92.7
3.	256	Regular mesh	713	64	69	73.8	78.6
4.	505	Irregular mesh	2786	163	158	175.9	193.5
5.	602	Plate non-mesh	3344	252	246	281.6	274.4
6.	200	Random	641	16	21	20.5	23.1
7.	400	Random	863	207	186	219.2	196.2

Table 1: Comparison of solution quality of ARM and SA on sample graphs

Comparison of running times of ARM & SA				
No.	Graph Characteristics		Running Times in sec.	
	$\|V\|$	Description	ARM	SA
1.	144	U-shaped mesh	15	5183
2.	192	Donut 8-point	20	6717
3.	256	Regular mesh	22	7019
4.	505	Irregular mesh	71	19491
5.	602	Plate non-mesh	89	22725
6.	200	Random	15	5609
7.	400	Random	34	10935

Table 2: Comparison of running times of ARM and SA on sample graphs

cessors," *IEEE Trans. on Computers,* Vol. C-36, May 1987, pp. 570-580.

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

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

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

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

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

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

[9] G.C. Fox and S. W. Otto, "Concurrent Computation and the Theory of Complex Systems," *Hypercube Multiprocessors 1986,* SIAM, Philadelphia, pp. 244-268.

[10] G. C. Fox and W. Furmanski," Balancing the Hypercube on a Neural Network," *Tech. report: Caltech Concurrent Computation Project,* Report #363, Sep. 1986. (to appear in *Journal of Supercomputing.*)

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

[12] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell Syst. Tech. J.,* Vol. 49, No. 2, 1970, pp. 291-308.

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

[14] O. Kramer and H. Muhlenbein,"Mapping strategies in message based multiprocessor systems," *PARLE 87* Vol. 1, Lecture Notes in Computer Science No. 258, Springer-Verlag, Berlin, Jun. 1987.

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

[16] V.M. Lo, "Task Assignment to Minimize Completion Time," *Proc. of the 5th Int'l. Conf. on Distributed Computing Systems,* May 1985, pp.329-336.

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

[18] H. Muhlenbein, M. Gorges-Schleuter and O. Kramer, " New solutions to the mapping problem of parallel systems: The evolution approach," *Parallel Computing,* Vol. 4, No. 3, Jun. 1987, pp. 269-279.

[19] J. Ramanujam, F. Ercal and P. Sadayappan, "Task Allocation by Simulated Annealing," to appear in *Proc. of the Int'l. Conf. on Supercomputing,* Boston MA, May 1988.

[20] P. Sadayappan and F. Ercal, "Cluster Partitioning Approaches to Mapping Parallel Programs onto a Hypercube," *Proc. Int'l Conf. on Supercomputing,* Athens, Greece, Jun. 1987.

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

[22] K. Schwan and C. Gaimon, "Automating Resource Allocation in the *Cm** Multiprocessor," *Proc. of the 5th Int'l. Conf. on Distributed Computing Systems,* May 1985, pp.310-320.

[23] C. Sechen and A. Sangiovanni-Vincentelli, "The TimberWolf Placement and Routing Package," *IEEE J. Solid-State Circuits,* Vol. SC-20, No. 2, Apr. 85, pp. 510-522.

[24] J. Sheild, " Partitioning concurrent VLSI simulation programs onto a multiprocessor by simulated annealing," *IEE Proceedings,* Part G, Vol.134, No.1, Jan. 1987, pp.24-28.

[25] C. Shen and W. Tsai, "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.

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

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