November 1988

Comparative Analysis of Hill Climbing Mapping Algorithms

David L. Smitley
Supercomputing Research Center

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-94.

Comparative Analysis of Hill Climbing Mapping Algorithms

Abstract

The performance of a parallel algorithm depends in part on how well the communication structure of the algorithm is matched to the communication structure of the target parallel system. The *mapping* problem is the problem of generating such a match algorithmically. Solving the mapping problem optimally for any non-trivial case is NP-complete. Therefore, a *heuristic* approach must be used to solve the problem. Although several heuristic algorithms to this problem have been developed, their performance has been evaluated on relatively few combinations of communication and processor structures. This paper extensively evaluates the performance of hill climbing mapping algorithms through simulation on communication structures representative of existing parallel algorithms and architectures. The motivations for our study are as follows: to establish the differences in performance between variations of the hill climbing heuristic; to determine the factors which affect the performance of hill climbing with respect to optimum; and to compare hill climbing to known optimum and non-optimum mappings to determine the effectiveness of hill climbing as a mapping heuristic.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-94.

COMPARATIVE ANALYSIS OF HILL CLIMBING MAPPING ALGORITHMS

David L. Smitley
and Insup Lee

MS-CIS-88-94
GRASP LAB 166

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

November 1988

Acknowledgements: This work was supported in part by NSF IRI84-10413-AO2, MCS-8219196-CER, NSF/DCR grants 8501482, 8410771, U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027 and Airforce F-49620-85-K-0018.

Comparative Analysis of Hill Climbing Mapping Algorithms

David L. Smitley

Supercomputing Research Center

4380 Forbes Blvd.

Lanham, MD 20706

Insup Lee

University of Pennsylvania

Dept. of Computer and Information Sciences

Philadelphia, PA 19146

ABSTRACT

The performance of a parallel algorithm depends in part on how well the communication structure of the algorithm is matched to the communication structure of the target parallel system. The *mapping* problem is the problem of generating such a match algorithmically. Solving the mapping problem optimally for any non-trivial case is NP-complete. Therefore, a *heuristic* approach must be used to solve the problem. Although several heuristic algorithms to this problem have been developed, their performance has been evaluated on relatively few combinations of communication and processor structures. This paper extensively evaluates the performance of hill climbing mapping algorithms through simulation on communication structures representative of existing parallel algorithms and architectures. The motivations for our study are as follows: to establish the differences in performance between variations of the hill climbing heuristic; to determine the factors which affect the performance of hill climbing with respect to optimum; and to compare hill climbing to known optimum and non-optimum mappings to determine the effectiveness of hill climbing as a mapping heuristic.

1. Introduction

One class of parallel processing systems in existence today consists of a number of processors, each with its own private memory. In addition, each of these processor-memory pairs are connected to a small number of other pairs in a fixed topology. In these systems, if two processor-memory pairs must share data, a message is constructed and sent through the interconnection network. Such a message must be forwarded through one or more intermediate processors in the network. This forwarding introduces delay and hence reduces the amount of speedup achieved.

A typical programming paradigm for this type of system consists of decomposing the problem into a number of communicating tasks that are assigned to individual processors. Since the interconnection network introduces delay on communication, one goal is to place the tasks on the processors so as to minimize the delay introduced by the interconnection network. This is referred to as the mapping problem.

For the mapping problem studied in this paper both the parallel algorithm and the processor system are represented as graphs. For a parallel algorithm, the nodes in the graph correspond to tasks and edges correspond to communication between tasks. For a parallel system as described above, nodes correspond to processors and edges to communication links. The mapping problem is then defined as finding a mapping from the nodes in the task graph to those in the processor graph so that a function defined on the two graphs is optimized. This function must be chosen so that optimizing its value minimizes communication delay.

Delay in a message passing systems consists of two principle components: delay due to forwarding and delay due to congestion. Delay due to forwarding results from the need to forward a message through one or more network links. Each traversal of a link adds to the time required to deliver a message. The amount of time required to traverse a link is a function of network congestion. If many messages arrive at an intermediate switching node and each want to traverse the same exit link, delay in the form of congestion is introduced.

In this paper we investigate delays due to message forwarding only. We take this approach for two reasons. First, in a system where the network is light to moderately loaded, it has been shown that forwarding delay dominates congestion delay [14]. Second, congestion effects are difficult to analyze given a static description of a parallel

task. Congestion at a particular network node depends on both *whether* and *when* messages arrive. However, using graphs to model parallel tasks, it can only be determined *whether* communication will occur; no timing information is available.

The relationship between message delay and task execution time depends not only on delay itself but also on the processor's mode of execution. For example, in a SIMD machine all communication starts at a fixed point in time and no further computation is performed until *all* messages have been delivered. In this case, reducing maximum message delay reduces execution time. This is in contrast to MIMD machines where computation can proceed with as few as *one* delivered message. In this case, reducing the *average* message delay reduces execution time. This paper concentrates on the latter case.

To define a function that relates a mapping between task and processor graph to average message delay, we introduce a value called *total expansion.* Assume that two nodes u and v are connected by an edge in the task graph. When u and v are mapped to the processor graph, messages flowing between them will flow between one or more intermediate processors. We say that the edge (u,v) has an *expansion* equal to the number of edges it traverses between the images of u and v in the processor graph. The *total expansion* of a task graph with respect to a processor graph is the sum of expansions for each edge in the task graph. Minimizing total expansion reduces the average message delay in a MIMD system where the time to communicate messages between processors is proportional to the length of the path.

One approach to the mapping problem has been that of finding mappings for specific task-processor graph combinations. Rosenberg and Snyder [12] were the first to take this approach when they developed lower bounds on total expansion for various task and processor graphs. They also presented specific mappings for several different task-processor graph combinations such as grids onto trees. This approach of constructing a specific mapping for a given task-processor combination has also been taken by other authors [6, 8, 5, 10]. These research efforts have resulted in optimum or near optimum mappings for many common communication structures onto typical processor graphs. However, for other task-processor combinations with no known mapping, determining an optimum mapping remains a difficult task.

As an alternative to manually constructing mappings for specific task-processor combinations, this paper investigates automated mapping generation for any task-

processor combination. That is, given a task graph T and processor graph G, find algorithmically a minimum total expansion mapping from T to G. This would allow the programmer to choose a communication structure that was natural for the problem without concern for how the communication structure is realized on a particular machine. In addition, if the topology of the processor system changes due to a fault, the mapping algorithm could be reapplied to the task graph to produce a mapping that would still exploit the remaining system configuration.

Finding an optimum mapping algorithmically is a NP-complete problem for any non-trivial optimization criteria [13]. Furthermore, fixing the processor graph often does not simplify the problem. For instance, solving the minimum total expansion problem to a line or ring graph is also NP-complete [13]. For these reasons heuristic mapping algorithms have been studied by several authors for a variety of optimization criteria [3, 7, 11]. These heuristic techniques are typically evaluated by executing the mapping algorithm on a small number of task-processor graph combinations. Often, the results are not compared to what is optimally achievable because the optimum solution is not known. Bokhari [3] attempted to deal with this problem by using his algorithm to map a graph onto itself. In this case the optimum mapping is known and the performance of the mapping algorithm can be determined. However, such experiments do not reveal the performance of the algorithm when presented with arbitrary task-processor graph combinations.

In this paper, we evaluate the performance of a heuristic mapping algorithm for a variety of task and processor graph combinations. We vary both the topology and size of the input graphs and determine the effects such changes have on the quality of the generated mapping. In addition, the total expansion of the mappings are compared to what could be achieved using known mappings, some of which are provably optimum. This allows one to determine how well a heuristic technique performs with respect to mappings that a programmer could specify directly.

As a heuristic, we have chosen hill climbing. The principle reason for this choice is its simplicity. One purpose of this paper is to provide a base set of experimental results for automated mapping algorithms. We feel that hill climbing is the most basic of heuristics and therefore provides a "least common denominator" to which more sophisticated heuristics can be compared as was done by Fukunaga, Yamada and Kasai [7].

With regards to the hill climbing algorithm, we use four variations to verify whether or not the results of Bokhari [3] (which showed that the best hill climbing strategy was steepest descent with multiple retrys) apply to generic task and processor graph combinations. As discussed in Section 6.1, this result does not seem to hold for general task-processor graph combinations.

The next section formally defines the mapping problem and describes the hill climbing heuristic. Section 3 describes the task and processor graphs used as test data. Section 4 then describes the set of experiments that were performed. In Section 5, the results of the experiments are presented. Conclusions are drawn from these results and presented in Section 6. Finally, Section 7 summarizes the results.

2. The Mapping Problem and a Heuristic Solution

The mapping problem as studied in this paper is defined as follows.

Given

A directed graph $T=\{V(T),E(T)\}, |V(T)|=n$

An edge weighing function w_T, $w_T:E(T)\rightarrow Reals$

An undirected graph $G=\{V(G),E(G)\}, |V(G)|=n$

Find

A bijective function m, $m:V(T)\rightarrow V(G)$ such that

$$\sum_{(u,v)\in E(T)} w_T((u,v))\, l_G(m(u),m(v))$$

is minimized. The term $l_G(u,v)$ is the minimum distance path between u and v in G.

The mapping problem as defined models the task graph as a weighted and directed graph. Directed graphs are used since many algorithms have asymmetric communication between two nodes. The directed edges are weighted proportional to the amount of data exchanged between the two communicating nodes. Communicating nodes that exchange large amounts of data have larger edge weights than nodes that exchange less data. For the sample parallel examples studied in this paper, we determine the exact number of data items exchanged and use this value as an edge weighting.

In contrast to the task graphs, the processor system is modeled as an undirected,

unweighted graph. This corresponds to a system where each link connecting two processors has the same capacity and is bidirectional. The mapping problem is that of finding a function from the task to processor graph so that the sum of *weighted* expansions is minimized. The expansion of an edge is also weighted to reflect the weight of the task graph edge.

2.1. The Mapping Algorithms

The algorithms studied in this paper are based on a combinatorical optimization technique known as hill climbing [16]. Hill climbing as a generic optimization technique starts with a guess for the solution. The current guess is then incrementally changed to yield the new guess. If the new guess improves the optimization criteria, the new guess becomes the current guess and the process is repeated; otherwise, the algorithm terminates.

The particular implementation used is as follows. To start, the task graph is randomly mapped to the processor graph. For each node u in the task graph, the algorithm performs a trial swap between u and all other nodes in the task graph. For each trial interchange, if the total expansion has decreased as the result of the interchange, then it is recorded. After all trial interchanges for node u are performed, if at least one interchange that decreases total expansion exists, one such interchange is selected and the mapping is updated accordingly. This paper investigates two ways of selecting interchanges: randomly and steepest descent. The former means choosing a swap from the set of all non-increasing swaps with equal probability. The latter means choosing the swap that decreases total expansion the most.

A pass consists of performing trial swaps for all pairs of nodes in the graph. If, during a pass, the total expansion has not decreased then the algorithm has entered a local minimum. At this point the algorithm can either terminate or perturb the current mapping and repeat the process of performing trial interchanges. This paper also investigates both methods of handling local minima. For the case where the current mapping is perturbed, the perturbation consists of performing n randomly chosen interchanges. Additional passes are then performed until no improvement during the current pass is made.

We note here that the algorithm starts with an arbitrarily chosen permutation. For the experiments described in Section 5, the starting permutation is randomly generated.

Results by Lee and Aggarwal [9] suggest that the performance of a hill climbing heuristic can be improved by choosing a good initial mapping. The effect of the initial mapping is not studied in this paper.

3. The Task Graphs

A variety of task graphs were used as test input. The graphs used are categorized as: 1 and 2-dimensional, tree and Fast Fourier. For each category we specify a parameterized generic graph and edge weighting function. One of the parameters specifies the number of nodes in the graph. Another parameter sets the number of edges per node (degree). A third parameter characterizes the amount of traffic on each edge. This weighting parameter is determined by considering a specific application that uses the particular graph. We now present a detailed specification of each generic task graph.

3.1. 1-Dimensional

Given a set V consisting of n nodes labeled consecutively from 1 to n, define a set of edges E parameterized by d, $1 \le d < n$ as follows:

$$E = \{(x,y): 1 \le x \le n,\ max(1,x-d) \le y \le min(x+d,n)\}\ 1 \le d < n.$$

This resultant graph $G=(V,E)$ is a 1-dimensional graph, where each node communicates with its d nearest left and right neighbors assuming no wrap around. Parallel algorithms that exhibit this structure include 1-dimensional convolution and the taking of 1-dimensional derivatives. To define a parameterized edge weighting function, we consider 1-dimensional convolution as the driving application. The parameters of a 1-dimensional convolution are m, the number of points to convolve, and k the size of the convolution kernel. Assume that m is an integer multiple of n, the number of processors. Assign sequentially m/n data points to each processor. Each processor must then communicate with its $d=\left\lceil (\lfloor k/2 \rfloor\ n)/m \right\rceil$ nearest left and right neighbors.

A way of specifying a weighting function is to assume that all neighboring points must be transferred for each convolution of a local data point. This would be the case if individual processes within the processor were responsible for a single data point. Consider a data point whose index within processor i is z. Assuming k is odd, $m/n-z$ of the points needed to perform the convolution of point z are in processor i and

$OPP=(k-1)/2-(m/n-z)$ points are in $B=\lceil OPP/(m/n) \rceil$ successive left and right processors. If the difference between processor i and j is given by:

$$|i-j| \leq \lfloor OPP/(m/n) \rfloor$$

then to convolve point z requires a fetch of (m/n) data points from processor j. If

$$\lfloor OPP/(m/n) \rfloor < |i-j| \leq B$$

then $OPP-(m/n)\lfloor OPP/(m/n) \rfloor$ points need to be fetched. Thus, the weight of edge (i,j) can be determined by summing the number of fetches needed from block j for each point z, $1 \leq z \leq m/n$ in block i.

3.2. 2-Dimensional

Given a set V consisting of $n=w^2$ nodes label the nodes using ordered pairs (x,y) such that $1 \leq x,y \leq w$. On these ordered pairs of nodes define a set of edges E parameterized by d as follows:

$$E = \{((x,y),(x',y'))\}$$

$$1 \leq x,y \leq w,\ max(1,x-d) \leq x' \leq min(x+d,w),\ max(1,y-d) \leq y' \leq min(y+d,w),\ 1 \leq d < w.$$

The resultant graph $G=(V,E)$ is a 2-dimensional analog to the 1-dimensional graph described above. One use for this graph is the solution of 2-dimensional differential equations using a cross shaped stencil. Another use is matrix multiplication.

The weighting function used is derived for the solution of two dimensional differential equations using a cross shaped stencil. The parameters to the problem are the r,s dimensions of the input data and the t,u dimensions of the stencil. Assuming that $r=s$ and r is an integer multiple of w, the data can be divided into w^2 regions of $(r/w)^2$ data points. These regions are labeled (x,y) and assigned to processors labeled (x,y). If $t=u$ each processor must then communicate with its $d=\lceil (\lfloor t/2 \rfloor n)/w \rceil$ nearest up, down, left and right neighbors. Note that when $r=s=t=u$, this graph also models a way of performing matrix multiplication.

A weighting function similar to the one given for the 1-dimensional graph can then be defined. The function is identical with the exception that the weights for the horizontal edges are multiplied by the number of points per processor in the vertical direction

and similarly that the weights on the vertical edges are multiplied by the number of points per processor in the horizontal direction.

3.3. Fast Fourier Transform

On a set V of $n = 2^p$ consecutively labeled nodes, define the the following set of edges:

$$E = \{(x,y): 0 \le x < n, y = x \oplus 2^i \; 0 \le i \le p\}$$

where $\oplus$ is the exclusive or function. The resulting graph $G=(V,E)$ is the Fast Fourier Transform or "butterfly" pattern. The principle use of this graph is to perform Fast Fourier Transforms but it also serves as an underlying communication graph for matrix transpose.

If this graph is used to perform FFTs on 2^q, $p \le q$, data points, every task must send 2^{q-p} data points to each task to which it is connected. In this case, the weighting function is a constant.

3.4. Tree Graphs

Another class of generic communication graphs we choose is d-ary trees. In the n node, d-ary trees that we use, all interior nodes have degree d. Furthermore, we construct the tree so that the maximum number of leftmost leaf nodes also have degree d. These graphs can be used to perform associative operations such as addition in parallel. Here, the values to be combined associatively are input at the leaves. The values are then combined at the roots of the respective subtrees and then passed to the next higher sub-tree. This process continues until all values reach the root of the tree where they are combined a final time to produce the result. Used in this manner, data is only passed up the tree and therefore the graphs used as test data have directed edges from leaf to root nodes. For the purposes in this paper we will weight the top $n/2-1$ edges in the with a weight of 2 and the remaining edges with a weight of 1.

3.5. Processor Graphs

In addition to using several different task graphs as input data, different processor graphs were also used. The graphs used were a 4 and 8-nearest neighbor, 2-dimensional toroid, a hypercube and a 7-tree. These topologies were chosen since they have been either proposed in the literature or exist as topologies in currently existing machines. All edges in the graphs are defined as undirected and unweighted. This models physical systems that are interconnected with bidirectional links with equal capacity. A brief description of each of the graphs is as follows:

The four nearest neighbor, two dimensional toroid consists of $n=w^2$ nodes labeled as ordered pairs (i,j). There exists an edge between each node (i,j) and nodes $(i\pm1 \ mod \ w, \ j)$, $(i, j\pm1 \ mod \ w)$. The 8-nearest neighbor toroid has an additional four edges given by the four combinations of $(i\pm1 \ mod \ w, j\pm1 \ mod \ w)$. The hypercube is the same graph as the FFT task graph described previously. The graph of the 7-tree is identical to the 7-tree description given in the task graph section with the exception that all links in the tree are considered to be unweighted and bidirectional.

4. The Experiments

For each of the generic task graphs, parameters were chosen to specify the graphs used for the experiments. One goal of the experiments was to determine the effect of task and processor graph size on the performance of the hill climbing algorithms. For this reason task graphs of n=16,64 and 256 nodes were used. These sizes enabled us to determine the relative change in performance with respect to a fourfold increase in graph size.

For the results described in Section 5.1, various *densities* of task graphs were used. For the 1-dimensional graphs m and k were chosen so that d, the number of nearest neighbors, would take on the values 1, 2, 4 and 8. To accomplish this, m was set to $10n$ and k to 11,31,71,151. These parameters were also used for the 2-dimensional graphs with the exception of the d=8 case. Instead of using d=8 as a test case we used $d=\sqrt{n}$. This models the communication requirements of matrix multiply. For the tree graphs, values of 2 and $n-1$ were used for d.

Edge weights were chosen according to the specific algorithm that the task graph was chosen to model. For the 1 and 2-dimensional convolution graphs, the weighting

functions described in Sections 3.1 and 3.2 were used. For the 2-dimensional matrix multiply graph, the weighting function described in Section 3.2 assigns the same weight to each edge. For the FFT graph, the weighting parameter q was set to n so as to give edges with a weight of one. For the associative graphs, the arbitrarily chosen edge weights described in Section 3.4 were used.

Each task graph of size n was mapped onto each processor graph of size n using four variations of the hill climbing algorithm. The four algorithms are derived from the four combinations of random/steepest descent move generation together with jump/no jump out of local optima. We label these algorithms as follows: algorithm 1 for random/no jump, algorithm 2 for steepest descent/no jump, algorithm 3 for random/jump and algorithm 4 for steepest descent/jump. For each task-processor combination, the four algorithms were run a number of times, each time with a different initial mapping and random number seed. The algorithms were run 50 times for the n=16 and 64 node experiments, and 10 times for the 256 node experiments.

5. Results

To determine the performance of the hill climbing algorithms, the total expansion of the generated mappings are compared in several different ways. First, in Section 5.1, the total expansion is compared to a lower bound on total expansion. These numbers are used to compare the performance of the hill climbing algorithms used. Second, in Section 5.2 the generated total expansion is compared with that of known mappings. The procedures for generating these mappings are also described in Section 5.2. In some instances, the mappings can also been shown to be optimal. Comparisons to these mappings are then used in Section 6.2 to determine the quality of a mapping found by hill climbing.

5.1. Results with Respect to Lower Bound

To determine the performance of the hill climbing algorithms, we compared the total expansion of the generated mappings to a lower bound. A lower bound TE_{lb} is given by the sum of weights over all edges in the task graph. This corresponds to a mapping in which each edge in the task graph has an expansion of one. Formally, this lower bound is defined as follows:

$$TE_{lb}(T) = \sum_{(u,v)\in E(T)} w_T(u,v)$$

Let $TE_{gen}(T,G,i)$ denote the total expansion of a mapping from T to G using hill climbing algorithm $i, 1 \le i \le 4$. Table 1 gives the average and maximum values of $TE_{ratio}(T,G,i) = TE_{lb}(T)/TE_{gen}(T,G,i)$ averaged over all tasks T for each processor graph G. The values in the average columns were obtained by first averaging the values of $TE_{ratio}(T,G,i)$ over 50 (10 for n=256) runs of algorithm i. Each run of an algorithm was conducted using a different starting permutation and random number seed. These values were then averaged for each processor graph G over all task graphs T. The maximum columns represent the maximum value of $TE_{ratio}(T,G,i)$ observed over all runs for each specific T and G.

In addition to determining the performance of the hill climbing algorithms with respect to a lower bound, we also counted the number of passes performed. These values were averaged across all task and processor graphs with n nodes and are given in Table 2. Recall that a pass consists of performing a trial swap for each pair of nodes in the task graph. This requires n^2 swaps. Therefore, the total number of swaps performed by each algorithm can be calculated by multiplying the entries in Table 2 by n^2. The values in Table 2 show that the number of passes for each variation of hill climbing grows at an approximate rate of $O(\log n)$. Hence, the number of trial swaps grows at an approximate rate of $O(n^2 \log n)$.

5.2. Results with Respect to Known Mappings

The values given in Table 1 are with respect to a lower bound on total expansion and are averaged across all task graphs. This lower bound allows one to compare the relative performance of each algorithm. Furthermore, it allows one to compare the efficacy of the various processor topologies at supporting the range of task graphs. For instance, at 256 nodes the results show that the hypercube provides the best match to the task topologies followed by the 8-toroid, the 4-toroid and then the 7-tree. The results given in Table 1 could also be used to compare hill climbing to other more sophisticated heuristics.

The results given in Table 1, however, do not provide any information as to how well hill climbing performs when mapping a specific task to a specific processor. To

	Average				Maximum			
	1	2	3	4	1	2	3	4
16 Nodes								
4t	0.712	0.732	0.750	0.771	0.800	0.800	0.800	0.800
8t	0.916	0.927	0.933	0.936	0.944	0.944	0.944	0.944
hc	0.717	0.736	0.753	0.767	0.800	0.800	0.800	0.800
7t	0.533	0.575	0.557	0.582	0.587	0.587	0.587	0.587
64 Nodes								
4t	0.537	0.537	0.554	0.552	0.592	0.589	0.604	0.603
8t	0.700	0.699	0.714	0.712	0.743	0.742	0.745	0.743
hc	0.623	0.628	0.643	0.646	0.687	0.708	0.708	0.706
7t	0.440	0.468	0.453	0.478	0.481	0.492	0.481	0.496
256 Nodes								
4t	0.425	0.411	0.438	0.423	0.452	0.433	0.458	0.441
8t	0.560	0.539	0.572	0.554	0.590	0.567	0.593	0.577
hc	0.574	0.568	0.586	0.577	0.598	0.585	0.604	0.592
7t	0.379	0.382	0.388	0.391	0.397	0.401	0.403	0.403

4t: 4-toroid
8t: 8-toroid
hc: Hypercube
7t: 7-tree

Column 1: Random choice, No jump out of local minima
Column 2: Steepest descent, No jump out of local minima
Column 3: Random choice, Jump out of local minima
Column 4: Steepest descent, Jump out of local minima

Table 1. Hill Climbing Algorithm Performance

determine such a relationship, comparative mappings are needed. We now describe size parameterized mappings for a subset of the experiments described in Section 5. Not all of the mappings described are optimum, but are presented to compare how well hill climbing does with respect to a known mapping for a specific task-processor combination.

5.2.1. Optimum Mappings

An embedding is a mapping for which each edge in the task graph has expansion one. Embeddings exist for a number of the task-processor combinations used as test data. Consider the 1-dimensional, 1-nearest neighbor graph. This graph is simply a line.

n	1	2	3	4
16	3.418	2.656	8.529	6.328
64	7.913	4.874	20.914	13.037
256	15.089	8.719	40.740	23.956

n - number of nodes
Column 1 - Random choice, No jump out of local minima
Column 2 - Steepest descent, No jump out of local minima
Column 3 - Random choice, Jump out of local minima
Column 4 - Steepest descent, Jump out of local minima

Table 2. Average Number of Trial Swaps

Therefore, there exists an embedding from this task graph to any processor graph that has a Hamiltonian circuit. It can easily be shown that Hamiltonian circuits exist in the 4 and 8-toroids together with the hypercube. Similarly, the 2-dimensional, 1-nearest neighbor graph is equivalent to a 4-nearest neighbor mesh and thus has an embedding in both 4 and 8-toroids. Furthermore, it has been shown by several authors [8, 5] that four nearest neighbor meshes embed in hypercubes. Finally, there exists an embedding between the FFT and hypercube graphs since the task and processor graphs are the same. The 16 node FFT can also be embedded in both the 4 and 8-toroid.

In the case of the $n-1$ tree task graph, there does not exist an embedding onto any of the processor graphs. However, there does exist a provably optimum mapping between a $n-1$ tree (star graph) and each of the processor graphs. Such a mapping is found by first determining the node in the processor graph with the smallest total distance between itself and the other nodes in the processor graph. The total distance between a node u and all other node is given by

$$d(u,G) = \sum_{v \in G} l_G(u,v)$$

where $l_G(u,v)$ is the minimum distance between nodes u and v in G. Once the node m such that $d(m,G) \leq d(u,G)$ for every $u \in G$ is identified, the root of the star is mapped to m.

In a *symmetric* graph, the sum of distances between a node u and all other nodes is equal regardless of what node u is chosen. For symmetric graphs such as toroids and hypercubes, any node in the graph can serve as m. Now, if the star is unweighted, the remaining nodes of the star can be mapped arbitrarily. Otherwise, the nodes v of the processor graph $v \neq m$ are sorted in ascending order according to their distance from m. Similarly, the nodes of the task graph are sorted in descending order according to the weight of the edge connecting that node to the root. A mapping is then made in order between the nodes in the task graph with heavier edge weights to the nodes in the processor graph with shorter distances from m. This mapping can easily be shown to have minimum total expansion.

5.2.2. Non-optimum Mappings

We now consider non-optimal task-processor graph mappings. Rosenberg and Snyder [12] presented a method of mapping n^d node, d-dimensional arrays into the *leaves* of complete 2^d-ary trees. We took their approach and expanded it to handle the problem of mapping an array to the *nodes* of a 7-tree by first mapping the d-dimensional array to the leaves of a 2^d-ary tree and then mapping those leaves into the nodes of a 7-tree. To map the leaves of a 2^d-ary tree to the nodes of a 7-tree, we first label the nodes of the 7-tree using post order traversal (traverse leaves of root followed by the root). The leaves of the 2^d-ary tree (to which the array has been mapped) are then assigned sequentially (from left to right) to the labels of the 7-tree obtained by the post order traversal. Rosenberg and Synder's method in conjunction with the leaf to node mapping thus defines mappings for the 1-dimensional, 1-nearest neighbor graph (a 1d array) and also the 2-dimension, 1-nearest neighbor graph (a 2d array) onto the 7-tree.

For the binary tree associative task graph, we used the shape grammar mapping technique described by Bailey and Cuny [1] to map the binary tree to both the 4 and 8-toroid processor graph. To map the binary tree graph onto a 7-tree processor graph, we labeled the nodes of both graphs in preorder traversal order. The nodes of the binary tree in preorder order were then sequentially mapped to the nodes of the 7-tree in preorder order. To map the binary tree to a hypercube, Deshpande and Jenevein [6] showed how to embed a modified $n = 2^k - 1$ complete binary tree in a 2^k node hypercube. The embedding was accomplished by first adding a node between the root of the binary tree and either its left or right son. This 2^k node graph then embeds directly onto a hypercube.

The binary trees that were used in the experiments described in Section 5 consisted of a 2^{k-1} node binary tree with one additional leaf node added to the leftmost leaf of the complete binary tree. This node is mapped to the extra node added by Deshpande and Jenevein, and the remaining nodes are mapped as described in their paper.

To map the FFT task graph onto the 4-toroid graph, we first label the nodes of the FFT task graph in binary representation as described in Section 3.3. We take the p bit binary address $l_1, \cdots l_p$ and divide it into two separate labels $L_0=l_1, \cdots l_{p/2}$ and $L_1=l_{p/2+1},...,l_p$. Then, the binary gray code representation [2] is calculated for both L_0 and L_1. A node in the task graph is then mapped to $(bgr(L_0),bgr(L_1))$ where bgr is the reflected binary gray code and the 2-tuple (i,j) addresses the processor graph node in row i, column j. We use this mapping for both the 4 and 8-toroid.

The last mapping considered is that of a 1-dimensional, 2-nearest neighbor task graph to the 4 and 8-toroid processor graphs. For this task graph the $(u,u\pm1)$ edges form a line graph. A mapping between this line graph and a Hamiltonian circuit in the 4-toroid processor graph gives the $(u,u\pm1)$ edges an expansion of one while the $(u,u\pm2)$ edges an expansion of two. Given that the number of nodes in a 1-dimensional, 2-nearest neighbor task graph has an intergal square root which is also divisible by four, there exists a mapping onto an 8-neighbor toroid with smaller total expansion than the 4-toroid mapping. The mapping is given by sequentially partitioning the sequentially labeled task graph into sets of size $2\sqrt{n}$. The sets are labeled $1 \cdots n/2\sqrt{n}$. For the odd numbered sets $1 \le k < n/2\sqrt{n}$, the nodes are assigned in downward blocks of four starting at row 1, column $2k-1$. Labeling in downward blocks of four means sequentially assigning task nodes to processors (i,j), (i, j+1), (i+1, j) and (i+1, j+1) where i is the row number and j the column number of a processor. For the even numbered sets $2 \le k \le n/2\sqrt{n}$, the nodes are assigned in upward blocks of four starting at row $\sqrt{n}$, column $2k-1$. Upward block of four assignment sequentially maps task nodes to processors (i,j), (i, j+1), (i-1, j), (i-1, j+1).

Each of the above described mappings were applied to their respective task-processor graph combination and the total expansion of the mapping calculated. Each of the values were in turn divided into the average and maximum total expansion found by hill climbing algorithm 1 (random moves and no jump out of local optimum). The results of these calculations are given in Table 3.

	Average				Maximum			
Topology	4t	8t	hc	7t	4t	8t	hc	7t
16 Nodes								
1d-1	0.87	1.00	0.89	0.94	1.00	1.00	1.00	1.14
2d-1	0.81	0.98	0.83	1.03	1.00	1.00	1.00	1.09
star	0.98	0.96	0.98	0.85	1.00	1.00	1.00	1.00
binary	1.17	0.99	1.07	0.99	1.25	1.00	1.13	1.09
FFT	0.74	0.98	0.75	*	1.00	1.00	1.00	*
1d-2	0.96	0.99	*	*	1.00	1.02	*	*
64 Nodes								
1d-1	0.77	0.94	0.83	0.83	0.88	1.00	0.93	0.93
2d-1	0.61	0.79	0.70	1.01	0.70	0.88	0.76	1.07
star	0.99	0.99	0.99	0.96	0.99	0.99	1.00	1.00
binary	1.08	1.08	0.89	1.13	1.20	1.14	1.01	1.28
FFT	0.78	0.99	0.61	*	0.92	1.01	0.81	*
1d-2	0.84	0.90	*	*	0.90	0.98	*	*
256 Nodes								
1d-1	0.69	0.85	0.81	0.78	0.75	0.89	0.87	0.80
2d-1	0.47	0.61	0.63	0.99	0.50	0.67	0.66	1.01
star	0.99	0.99	0.99	0.99	0.99	0.99	1.00	1.00
binary	1.07	0.79	0.87	1.13	1.17	0.83	0.91	1.18
FFT	0.78	0.99	0.50	*	0.82	1.01	0.54	*
1d-2	0.73	0.77	*	*	0.77	0.8	*	*

1d-1: 1-dim., 1-nearest neighbor
2d-1: 2-dim., 1-nearest neighbor
star: star graph
binary: binary tree
FFT: Fast Fourier Transform
1d-2: 1-dim., 1-nearest neighbor

4t: 4-toroid
8t: 8-toroid
hc: Hypercube
7t: 7tree
*: no mapping

Table 3. Ratio of Hill Climbing to Known Mapping

6. Analysis

Given the data collected in Section 5.1 and the known mappings described in Sections 5.2 and 5.3, we now analyze the performance of the hill climbing algorithms. We first analyze the relative performance of the four different hill climbing algorithms. This is accomplished by comparing the total expansion of the mappings generated together with the number of trial swaps performed. We then analyze the performance of hill climbing with respect to the set of known mappings. Based on these comparisons, we evaluate the quality of the mappings found by the hill climbing algorithms.

6.1. Performance with Respect to Hill Climbing Algorithm

The columns in Table 1 show that the differences in performance across the four variations of the hill climbing are relatively small. Comparing the performance of algorithm 1 to algorithm 2, no consistent difference in TE is observed across graph size or processor topology. Similarly, no consistent difference in TE is observed between algorithm 3 and 4. This implies that steepest descent move selection does not generate mappings with smaller total expansion than does random move selection. However, Table 2 shows that steepest descent makes between 23 to 43 percent fewer trial swaps than random move selection. Furthermore, the difference in the number of trial swaps increases as the size of the input graphs increase. Therefore, while steepest descent does not improve the quality of the mapping found, it finds mappings more expediently than random move selection.

Whereas there are no consistent differences between steepest descent and random move generation, there exists a consistent difference between the corresponding versions of the hill climbing algorithm that either stay or attempt to move out of local optima. This can be seen by comparing the average results of algorithms 1 and 3 (Table 1, columns 1 and 3) and also algorithms 2 and 4 (Table 1, columns 2 and 4). A consistent difference is also observed for the maximum values.

These results imply that moving out of a local optimum causes the algorithm to finally terminate in local optima with smaller total expansion. However, this small increase in average case performance comes at the expense of time. Jumping out of local optima increases the number of passes (and hence total trial swaps) from between 2.4 to 2.7 times that required if the algorithm terminates at the first local optima found (Table

2). Furthermore, the increase in number of swaps resulting from jumps out of local optima increase with the size of the graph.

6.2. Performance with Respect to Known Mappings

In this section we discuss the performance of hill climbing with respect to known mappings. We first consider the optimum mappings described in Section 5.2 and then the non-optimum mappings described in Section 5.3. The optimum mappings include the 1 and 2-dimensional, 1-nearest neighbor task graphs onto the 4-toroid, 8-toroid and hypercube and also the FFT onto the hypercube. For these task-processor graph combinations, at least one of the 50 runs of the hill climbing algorithm found the embedding for 16 node graphs. However, for the 64 and 256 node graphs, the hill climbing algorithm never found an embedding. Furthermore, the quality of the mapping that was found decreased with increasing graph size. For example, the mappings found between the FFT and hypercube (Table 3) increased the average total expansion from 4/3 to 2 times optimum as n increased.

Fixing the task graph and examining the quality of the mapping for a range of processor graphs shows that the performance of hill climbing depends on the chosen processor graph. Consider the mapping results for the 1 and 2-dimensional, 1 nearest neighbor task graphs onto the hypercube, 4-toroid and 8-toroid processor graphs. For the 64 node graphs, the relative order of mapping from worst to best is for the 4-toroid, hypercube and 8-toroid. This order holds for both the 1 and 2-dimensional task graphs. This order corresponds to the same ordering obtained if the processor graphs were ordered according to their average internode distance. The same result also holds for the 16 node graphs. Hence, the results for 16 and 64 nodes suggests that the quality of mapping found (when an embedding exists) depends on the average internode distance of the processor graph.

However, the results for the 256 node experiments show that the relative mapping performance order for the 1-dimensional task graph is 4-toroid, hypercube and 8-toroid; whereas the order in terms of average internode distance is 4-toroid, 8-toroid and hypercube. This result suggests that mapping algorithm performance correlates with average internode distance only if the relative difference between average internode distance is large. For example, the relative difference in average internode distance between the 4-

toroid and hypercube is 100%, whereas the relative difference between an 8-toroid and hypercube is only 33%. For those processor graphs with similar average internode distance, the performance of the hill climbing algorithm appears to depend on the particular task graph being mapped. For instance, while the relative mapping performance order for the 1-dimensional task graph was hypercube then 8-toroid, the order for the 2-dimensional graph was 8-toroid then hypercube.

Similar to fixing the task topology, hill climbing performance can also be analyzed by fixing the processor graph topology. Fixing the processor topology to a hypercube and examining the mappings for the 1-dimensional, 2-dimensional and FFT tasks (each of which have an embedding in the hypercube topology), it can be seen that the relative order of mappings (with respect to an embedding) from worst to best is given by: FFT, 2-dimensional, 1-dimensional. These results hold for all graph sizes considered. Furthermore, the results hold when the processor topology is fixed to either a 4 or 8-toroid and the 1 or 2-dimensional task graph is mapped.

Examining this trend, it can be seen that the order in quality of mapping correlates directly with the number of edges in the task topologies. That is, the fewer edges a task graph has the better the mapping with respect to optimum. Furthermore, this effect increases as the size of the topology increases. For example, the relative average performance for the 64 node, 2-dimensional task onto a hypercube was 85% percent of the performance obtained for the 1-dimensional task. For the 256 node case the 2-dimensional task performance was only 77% of the 1-dimensional performance. Such a decrease in performance with respect to optimum is seen for all of the task-processor graph combinations for which an embedding exists.

Examining the maximum columns of Table 3 for the star graph shows that at least one execution of the hill climbing algorithm found a mapping within 99% of optimum for all processor graphs at all sizes. The reason for such performance (in comparison to the mappings found for task-processor combinations where an embedding exists) lies with the ease of mapping star graphs. The 4 and 8-toroid along with the hypercube graphs are examples of symmetric graphs. Any mapping of an unweighted star graph to a symmetric graph will have the same total expansion. If the star graph is weighted, heavily weighted edges need to be mapped close to the root of the star. It appears that the hill climbing algorithm optimizes effectively in this way for the simple weighted star

graphs used in the experiments. For the 7-tree which is not symmetric, the node u with the smallest sum of distances to other nodes must first be found by the mapping algorithm. Again, the results suggest that the hill climbing algorithm performs this function.

We now compare hill climbing to non-optimal mappings. Although the absolute quality of hill climbing cannot be determined, such comparisons are important since it shows how well general techniques such as hill climbing can perform with respect to specific mapping schemes. Furthermore, the changes in the ratios in Table 3 allow us to assess the relative performance of the mapping schemes as the size of the graphs increase. Recall that the ratios in Table 3 are calculated by dividing the TE of the given non-optimum mapping by the TE of the mapping generated by hill climbing. If this ratio decreases with increasing graph size, then it must be the case that the growth rate of TE for the given mapping with respect to optimum is slower than the corresponding rate for hill climbing.

Consider this ratio for the following task-processor combinations (Table 3):

- 1d-1, 2d-1 onto 7-tree
- 1d-2 onto 4-toroid, 8-toroid
- binary tree onto 4-toroid, hypercube

The ratios for each of these combinations decrease as the size of the graph increases. This implies that the mappings given in Section 6.2 are superior to those achievable with hill climbing, particularly as the size of the graph increases (with the exception of the binary tree to 4-toroid mapping which may not increase in performance past that of hill climbing).

For the following task-processor combinations the ratio of the TE of the non-optimum mapping to hill climbing TE does not monotonically decrease with increasing graph size:

- FFT onto 4-toroid, 8-toroid
- binary tree onto 7-tree

In these cases both the given mapping and the hill climbing mapping move away from the optimum solution at approximately the same rate. However, the mappings given in Section 5.2 outperform hill climbing for the FFT onto 4 and 8-toroid task-processor combination. For the binary tree to 7-tree, it appears that hill climbing performs as well or

possibly better than the given mapping depending on graph size.

For the binary tree onto 8-toroid, the growth rate of the ratio given in Table 3 does not consistently increase or decrease with an increase in graph size. The trend appears to be decreasing but further experiments would need to be conducted with larger graphs to determine the validity of this observation.

7. Summary

In this paper we have investigated four variations of the hill climbing heuristic to solve the mapping problem. We did this by empirically studying the algorithms' behavior for a variety of task-processor graph combinations. By comparing the mappings generated by hill climbing to both optimum and non-optimum mappings, we were able to determine the parameters that affect hill climbing performance. Furthermore, these results form a base set with which to compare the performance of more sophisticated heuristics.

In terms of the quality of mapping generated, each variation of the hill climbing algorithm had roughly the same performance. However, by using steepest descent, the number of trial swaps to termination was significantly reduced. Furthermore, attempting to find better solutions by jumping out of local optima significantly increases execution time with only a small improvement in the quality of solution.

The comparisons with respect to optimum embeddings showed two things. First, the performance of the hill climbing heuristic decreases with an increase in graph size. Hill climbing as implemented also suffers from a $O(n^2 \log n)$ running time growth rate. Thus, not only are the mappings for large graphs poor, but they also takes a relatively long time to find. For instance, on a single user Sun-3/280, mapping a 256 node, 2-dimensional, 1-nearest task graph onto the 8-toroid took 2 minutes, 40 seconds using the fastest hill climbing algorithm (steepest descent/no jump). Second, when an exact match exists between task and processor, the performance of the algorithm can be characterized both by the number of edges in the task graph and the average internode distance of the processor graph.

Compared to non-optimum mappings, hill climbing did not perform better and in most instances performed worse. Hence, for a particular task-processor combination the programmer has to decide whether hill climbing should be used. The alternative is to

write programs with communication structures that match the processor's communication structure.

In conclusion, hill climbing as a mapping technique is only useful for relatively small graphs. The results show that hill climbing comes within a factor of two of optimum for the moderate size graphs considered. The performance of hill climbing, however, decreases with increasing graph size. Alternative heuristics are thus needed for mapping tasks onto large parallel machines such as the CM-2 [15] which can have up to 65,536 processors. A refinement of hill climbing known as simulated annealing [4] is one possibility. However, any state space search technique will probably suffer from large running times due to the size of the state space.

References

[1] Bailey, D.A. and Cuny, J.E., "The Use of Shape Grammars in Processor Embeddings," COINS Technical Report A-86-23, Dept. of CIS, Univ. of Massachusetts (July 1986).

[2] Bitner, J.R., Ehrlich, G., and Reingold, E.M., "Efficient Generation of the Binary Relected Gray Code and Its Applications," *Communications of the ACM* **19** (Sept. 1976).

[3] Bokhari, S., "On the Mapping Problem," *IEEE Transactions on Computers* **C-30**(3), pp. 207-214 (Mar. 1981).

[4] Bollinger, S.W. and Midkiff, S.F., "Processor and Link Assignment in Multicomputers Using Simulated Annealing," *Proc. 1988 Int. Conference on Parallel Processing, Vol I - Architecture*, pp. 1-7, The Pennsylvania State University Press.

[5] Chan, T.F. and Saad, Y., "Multigrid Algroithms on the Hypercube Multiprocessor," *IEEE Trans. on Computers* **C-35**(11), pp. 969-978 (Nov. 1986).

[6] Deshpande, S.R. and Jenevein, R.M., "Scalability of a Binary Tree on a Hypercube," *Proc. 1986 Int. Conference on Parallel Processing*, pp. 661-668.

[7] Fukunaga, K., Yamada, S., and Kasai, T., "Assignment of Job Modules onto Array Processors," *IEEE Transactions on Computers* **C-36**(7), pp. 888-891 (July 1987).

[8] Johnsson, S.L., "Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures," *Journal of Parallel and Distributed Computing* **4**, pp. 133-172 (1987).

[9] Lee, S.-Y. and Aggarwal, J.K., "A Mapping Strategy for Parallel Processing," *IEEE Trans. on Computers* **C-36**(4), pp. 433-442 (April 1987).

[10] Ma, Y.E. and Tao, L., "Embeddings Among Toruses and Meshes," *Proc. 1987 Int. Conference on Parallel Processing*, pp. 178-187.

[11] Napolitano, L.M. Jr., ''Revisiting the Mapping Problem,'' Technical Report, Sandia National Lab, Livermore, CA (1986).

[12] Rosenberg, A.L. and Snyder, L., ''Bounds on the Costs of Data Encodings,'' *Mathematical Systems Theory* **12**, pp. 9-39 (1978).

[13] Rosenberg, A.L., ''Data Encodings and Their Costs,'' *Acta Informatica* **9**, pp. 273-292 (1978).

[14] Smitley, D.L., *The Utilization of Processors Interconnected with a Reconfigurable Network, Ph.D. Dissertation,* University of Pennsylvania (April 1987).

[15] Thinking, Machines. Corp, ''Model CM-2 Technical Summary,'' Technical Report HA87-4, Thinking Machines Corp., Cambridge, MA.

[16] Tovey, C.A., ''Hill Climbing with Multiple Local Optima,'' *SIAM Journal on Algebraic and Discrete Methods* **6**(3), pp. 384-393 (July 1985).