CMU-CS-82-136

A Study in Parallel Computation -- the Traveling Salesman Problem

Joseph Mohan

18 August 1982

Abstract: The Traveling Salesman Problem is solved on the Cm*, a multiprocessor system, using two implementations based on the branch and bound algorithm of Little, Murty, Sweeny and Karel. One of these implementations is synchronous and has a granularity that increases with the degree of parallelism, while the other is asynchronous and has a constant granularity. With increasing degree of parallelism, the first implementation requires increasing amount of computation to solve the problem, leading to a speedup that saturates at a low value. The second implementation requires nearly the same amount computation at all degrees of parallelism and has reasonable speedup characteristic. The difference between the speedup of this implementation and linear speedup is attributed to processors idling because of resource contention.

The Cm* multiprocessor project was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

This research was partly supported by the National Science Foundation under grant ECS-8120270.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the funding agencies or the US Government.

Table of Contents

List of Figures

1. Introduction

This paper presents computational results of solving the Traveling Salesman Problem on Cm*, a multiprocessor system of about fifty LSI-11 processors, and, based on these results, relates program performance in a parallel environment to some characteristics of parallel programs such as synchronism and granularity. Other issues in parallelization such as reliability and language characteristics are completely ignored. Broader computational issues get more attention than specific architectural features of the system. The approach is experimental and case study oriented, with Cm* serving as the experimental testbed and the Traveling Salesman Problem serving as the vehicle of exploration.

The Traveling Salesman Problem (TSP, for short) is this: given the costs of going from one city to another of N cities, find a minimum cost tour (circuit) visiting each of the N cities once and only once. Below is a little more mathematical statement of the problem. Consider a complete graph $G = (V, E)$, where V is a set of n vertices and E is the set of m edges where $m = (n-1)^2$. If c_{ij} is the cost associated with edge (i, j), $[c_{ij}]$ will be the cost matrix of the graph G; TSP is the problem of finding a Hamiltonian circuit of G with the minimum cost.

TSP has been and still is widely studied [Christofides 79, Bellmore 68, Hoffman 81] because it is an easy to state but computationally difficult problem. It has some direct application to real-life problems such as vehicle routing; more significantly, however, it underlies many other optimization problems such as sequencing and scheduling. It is NP-complete, making it computation-intensive and a good subject for this study of computational speedup.

There are several variations to the problem -- symmetric or asymmetric cost matrix, euclidean or non-euclidean distances as costs, and exact or approximate solution. We chose to find the exact solution to the asymmetric non-euclidean TSP. Asymmetric non-euclidean TSP is the most general of all variations. *Asymmetric* means that, in general, c_{ij} does not equal c_{ji} for the given cost matrix. *Non-euclidean* means that the costs are arbitrary nonnegative numbers (integers, for us) and we can not expect them to obey any special property such as that of triangle inequality. Trying to find an exact solution rather than an approximate one helps us to focus our attention on issues of speed and avoid other issues such as accuracy of solution.

Dynamic programming [Karp 62] and many variations of branch and bound are the two most popular techniques for solving TSP exactly. For the TSP, good branch and bound algorithms are more efficient than dynamic programming ones -- branch and bound algorithms have the same worst-case complexity as dynamic programming ones, but typically take much less time to solve a problem

because of the bounding heuristics [Nilsson 71]. We decided to use a branch and bound algorithm for our experiment because branch and bound algorithms are a class of algorithms of wide and general applicability [Lawler 66, Mitten 70]. Any general results that we can obtain about this class of algorithms will be of interest to the computing -- both core and application -- community.

There are many branch and bound algorithms for the TSP [Little 63, Karp 70, Karp 71]. Our programs are based on the branch and bound algorithm of Little, Murty, Sweeny and Karel [Little 63] (we will refer to this algorithm as the LMSK algorithm hereafter). The LMSK algorithm is simple and easy to implement. (A variation of this algorithm is elaborated in [Horowitz 78].) The two programs lie on significantly different points in the implementation spectrum so that a comparative analysis of their performance will shed some light on the relation between performance and program characteristics.

2. Computational Environment

The experiments were conducted on the multiprocessor system Cm* [Swan 78] running the operating system StarOS [StarOS 81]. Cm* is a hierarchically organized multiprocessor system built at Carnegie-Mellon University. StarOS is a capability-based multiprocessor operating system, built to be a testbed for experimentation in parallel processing.

Highlights of the Cm* system that are relevant to the study are listed here:

> The Cm* system currently consists of five clusters of about 10 Cms each; each Cm consists of an LSI-11 processor with some memory.

> Each cluster has one high performance mapping processor, called KMap, which maps addresses from Cm address space to global address space and handles communication across clusters. KMaps create a uniform address space making any individual word in the system accessible to all the processors.

> Some addresses are local to a Cm and can be accessed without having to go through the KMap and hence without having to pay an extra access cost. Typically, accesses to the memory of another Cm in the same cluster costs about three times the access cost to the Cm's own (local) memory and accesses to remote clusters cost about twelve times [Jones and Gehringer 80].

> Accesses across clusters are achieved via the KMaps over a common bus called an intercluster bus (though there are two such busses, only one of them is used by the operating system).

Some pertinent features of the operating system StarOS are given below:

> Capability based addressing and protection of objects.

> Programs organized as modules and functions.

> Communication between processes through shared objects or mailboxes.

> One Object Manager per cluster; an Object Manager manages the memory in its cluster and fabricates objects out of it whenever a user requests.

> A typical process consists of a few code objects, a stack and local object, and some shared and some private data objects.

> A process can be made to execute on a particular Cm only, by assigning that process to what are known as private RunQueues. The default is for a process to run on any free Cm in a particular cluster.

> It is possible to localize objects to specific Cms to improve execution times; *localize* here means forcing an object to be mapped to the local memory of a Cm resulting in faster access to that object by a process running on that Cm.

> KMaps, in addition to their usual mapping functions, offer a primitive lock mechanism as an indivisible increment word (IncrWord) operation.

> The system has a clock with a clock-tick equal to 2 micro-seconds.

The programs were written in Bliss-11 [Wulf *et al.* 70] and cross-compiled and linked [Levin *et al.* 76] on a DEC-10 for execution on the Cm*.

3. LMSK Algorithm

The algorithm works by partitioning the set of tours into smaller and smaller subsets, finding a *lower bound* on tour cost of each of the subsets, and using these bounds to guide further partitioning of the tours until we get a single tour whose tour cost is less than or equal to the lower bounds for all the other subsets.

Lower bound for a tour subset with given cost matrix is computed using a matrix *reduction* operation. If a constant, h, is subtracted from any row or column of a cost matrix, there is a one to one correspondence between the tours under the new matrix and the tours under the old matrix, with the cost of the tours under the new matrix less by h; the optimal tour remains the same. A matrix with at least one zero in each row and one zero in each column is called a *reduced matrix*. If a tour had a cost $z(t)$ under a given cost matrix and a cost $z_1(t)$ under the corresponding reduced matrix and if h is the sum of the constants used in making the reduction,

$$z(t) = h + z_1(t).$$

Assuming nonnegative costs, h is the lower bound of all tours under the original matrix.

Partitioning of the tours can be represented by the branching of a state space tree.The root node of the tree corresponds to the set of all tours. One of its children will be an *include* node and corresponds to all those tours of the parent node that included a particular edge *(i, j)* selected at this level; the other node (the *exclude* node) corresponds to those tours that exclude that edge. The include/exclude edge *(i, j)* is selected to maximize the possibility of the best tour *not* being contained in the exclude subset; this is because inclusion of an edge reduces the size of the problem by one, which takes the computation one step nearer to the solution, while exclusion of an edge does not. The tours that exclude the edge *(i, j)* must include at least the sum of the costs of the smallest element in row *i* and the smallest element in column *j*, *after* excluding c_{ij}. This sum will be referred to as *theta(i, j)*. We choose the next include/exclude edge to be the one with the largest *theta(i, j)*.

The cost matrix for a child node is derived from its parent node's matrix by modifying it to reflect the include/exclude decision made at that partitioning point. When an edge *(k, l)* is to be included, row *k* and column *l* are no longer needed and are deleted. Next the unique connected path starting at city *p* and ending at *m*, of which *(k, l)* is a part, is determined and c_{mp} is set to infinity to avoid generating subtours. (A subtour is a tour over a proper subset of the given set of cities.) When an edge *(k, l)* is to be excluded, c_{kl} is simply set to infinity.

The basic LMSK algorithm is presented below. For clarity of presentation, the program is in its usual single-process form. The adaptations that result in two different parallel programs will be discussed in section 4.

FindTour is the main routine of the program. After some initialization, it enters the main iterative loop, which consists of choosing an edge to include or exclude, setting up the children using the parent node's matrix appropriately modified, reducing the matrices to get their lower bounds, and selecting the node with the least lower bound for expansion next.

```
type    Cost = integer; /* edge or tour cost */
        N    =          /* node in state space tree */
             record
                Size    : integer; /* size of matrix for this node */
                W       : Cost;    /* lower bound on tour */
                k       : integer; /* FROM node of partitioning edge */
                l       : integer; /* TO node of partitioning edge */
                Include : boolean; /*  specifies whether selected edge
                                       (k, l) is to be included (true) or
                                       excluded (false) */
                Parent  : ↑N;           /* pointer to parent node */
                LeftChild : ↑N;         /* pointer to left child */
                RightChild : ↑N;        /* pointer to right child */
             end;
        M    = array [integer] of Cost; /* cost matrix */

var     Matrix : M;

procedure FindTour (CostMatrix: M; ProblemSize: integer);
    begin
    var NumberOfChildren : integer;
        Node             : ↑N;
        Z0               : Cost;

        NumberOfChildren := 2;
        Z0 := infinity;
        New(Node);                          /* Root node */
        Node↑.Size   := ProblemSize;
        Node↑.Parent := nil;

        repeat
            if Matrix does not correspond to Node then
                ReconstructMatrix(Node);

            ChooseEdge(Node);

            for I from 0 to NumberOfChildren-1 do
                SetUpChild(Node, I);
            if Node↑.Size = 2 then
                /* some tour has been found */
                if Node↑.W < Z0 then
                    begin
                    Z0 := Node↑.W;
                    SaveTour;
                    end;
            Order leaf nodes by W;
            Node := leaf node with smallest W;
        until Z0 leq Node↑.W;
    end;
```

The following routines are lower level routines; the reasoning behind them has been explained earlier in this section. The routines are presented here for the sake of completion and the casual reader may want to skip them.

```
procedure ChooseEdge (Node: ↑N);
    begin
        with Node↑ do
            Choose edge (k,l) such that
                theta(k,l) = max (theta(i,j)) for all (i, j)
                where theta(i,j) = (smallest cost in row i,
                                        omitting Matrix[i,j]) +
                        (smallest cost in column j, omitting Matrix[i,j])
    end;


procedure SetUpChild (Node: ↑N; ChildIndex: integer);
    begin
    var ChildNode : ↑N;

        New(ChildNode);
        SetLinks; /* set up the right links to put child node in tree*/
        if ChildIndex = 0 then
            begin
            /* (k, l) is the excluded edge */
            ChildNode↑.Include := false;
            ChildNode↑.Size := Node↑.Size;
            ChildNode↑.W := Node↑.W + theta(k,l);
            end
        else begin
            /* (k, l) is the included edge */
            ChildNode↑.Include := true;
            Delete row k and column l in Matrix;
            ChildNode↑.Size := Node↑.Size - 1;
            Find p = starting city and m = ending city of the path
                containing the edge (k,l) among paths generated by
                the committed city pairs of child node;
            Matrix[m, p] = infinity;
            Reduce(Matrix);
            ChildNode↑.W := Node↑.W + reducing constants;
            end;
    end;


procedure ReconstructMatrix(Node: ↑N);
    begin
        Matrix := CostMatrix;
        Node↑.W := 0;
        Traverse tree from Node to root doing the following:
            if edge (i,j) of an intervening node is an
                                        included edge then

                begin
                    Node↑.W := Node↑.W + Matrix[i,j];
                    delete row i and column j of Matrix;
                    for each path including the edge (i,j)
                        find starting city p and ending city m
                        and set Matrix[m,p] to infinity;
                end
            else /* edge (i,j) is an excluded edge */
                Matrix[i,j] := infinity;
```

```
        Reduce(Matrix);
        Node↑.W := Node↑.W + reducing constants;
    end;
```

4. Implementations

An algorithm may be adapted for parallel execution primarily in two ways. Input data may be divided up among many processes and the results of these processes integrated into a final solution. This approach is clearly inapplicable to the LMSK algorithm. On the other hand, potential parallel paths in the control flow of the algorithm may be explicated and computed by multiple processes. One technique that will generally yield a large degree of parallelism is unfolding a loop and letting multiple processes work on different iterations of the unfolded loop. This is the approach taken here. The two implementations unfold different loops of the algorithm. Implementation 1 unfolds the *for* loop that sets up child nodes (see routine *FindTour* in section 3), while implementation 2 unfolds the outermost (*repeat*) loop. This is the basic difference between the two implementations; the other differences will be discussed below.

4.1. Implementation one

In implementation 1, only the work of setting up child nodes is done in parallel; one process sets up one child node. The algorithm presented in section 3 decides on the inclusion of a single edge during each iteration; each interior node in the state space tree has two child nodes and, hence, the maximum parallelism that can be realized with this approach in the present form of the algorithm is 2. To gain greater parallelism at this level of parallelization, the algorithm is modified to decide on the inclusion of a greater number of edges in each major iteration. After this change, two processes set up the two child nodes corresponding to the inclusion and exclusion of a single edge, four processes set up nodes for two edges, and 2^N processes set up nodes for N edges (one node corresponds to one possible include/exclude selection of the N edges). The edges are selected with repeated applications of the same criterion (maximum theta) as was presented in the algorithm.

A *master* process controls the general progress of computation. It sets up several *slave* processes that set up the child nodes; it controls each of the iterations of the slaves, tells them which node to set up, decides when they are done, and performs bookkeeping for us.

Our decision to parallelize at the level of setting up child nodes mandates many other characteristics of this implementation:

- necessity to select increasing number of edges in each major iteration with increasing degree of parallelism

> need for a process (master) that does significantly more work than others and controls their execution, leading to a master-slave process structure; the slaves set up child nodes, while the master does the rest of the computation in *FindTour*

> need to wait for all child nodes to be set up before execution can continue; this leads to a synchronous control structure

> since the master process strictly partitions work between the slaves, no lock is necessary to ensure consistency of data.

4.2. Implementation two

Implementation 2 achieves its parallelism at the outermost iterative loop of the LMSK algorithm. All the work within the *repeat-until* loop of *FindTour*, including the loop-termination check, is done by all the processes. Each process, during each of its iteration, decides on one more edge and sets up two child nodes.

The characteristics of this program are less rigidly dictated by the parallelization scheme used. Some of its significant characteristics that differentiate it from implementation 1 are listed below:

> Parallelization is realized at a higher level of program control structure. Each process does all the work of expanding a node, setting up child processes, putting them in a work list of nodes ordered by their potential for leading to the solution, and choosing the next node for itself to expand.

> All the processes work asynchronously and independently. They are always busy doing useful computation, except for a short while in the very beginning when there are more processes than nodes and except when they find some global structure locked by some other processes for updating.

> There is no master-slave relationship among the processes. All processes are identical while solving the problem. (One process is distinguished and does the bookkeeping chores, but this does not affect our measurements.)

> The locking scheme is simple mutual-exclusion based on the StarOS instruction, IncrWord. If a structure is found locked, the process seeking to lock spins for a little while doing nothing and tries again. Two locks are used: one for the tree as a whole when setting up new child nodes into the work list and removing next node to be expanded from the list and another for maintaining global data such as a potential solution to the problem. Contention for these locks should be mild at low parallelisms (less than 16 here), since they are accessed at most once per iteration at random points in time by not too many processes for a short duration.

> Next node to be expanded is selected differently from the other implementation, although the criterion for selection is exactly the same -- the leaf node with the least lower bound on tour cost. All the leaf nodes in this implementation are linked in a list ordered by their potential cost with the least cost node at the front of the list. In the previous

implementation, we did not have an explicit ordered list, but simply traversed the tree from the current node to the root and back down to the next node; this was possible for that implementation because of the more controlled way in which the processes set up new child nodes.

Termination in an asynchronous algorithm has to be done carefully. When a process, *P*, finds that a terminal node (a node specifying a complete tour) has the least cost among all nodes currently available to it for inspection, it can not quit, nor can it decide that the problem has been completely solved. This is because other processes may be working on nodes that may lead to a better solution and these nodes are not accessible, for the moment, to process *P*. Implementation 2 uses a consensus-based termination scheme. Every process, on reaching a least-cost terminal node, increments a count and checks its value. If the count shows that all other processes also have come to the same conclusion, the problem is solved; otherwise the process waits a while and checks to see whether there is work to do and if there is, it decrements the count and continues to work.

4.3. Data Structures

The major data structures for our implementations are various cost matrices (N X N square matrices, where N is the size of the problem, that is, the number of cities) and a state space tree that records the current state of problem execution. Each process keeps a localized data object for the cost matrix and sets up the cost matrix corresponding to the node it is currently expanding on this object, when necessary. The nodes of the tree are set up in whichever memory area possible, giving preference, of course, to locations nearer (in the access hierarchy) to the Cm corresponding to the process creating a node.

The decisions on data structures are dictated by the requirements of the problem and the system environment. The problem dictates the tree and matrix structures. The per process matrices are easily localized when we start execution; the nodes of the tree are localized to the extent dynamically possible in the system. One matrix object is associated with one process rather than with each node. One matrix for each process can be easily accommodated, but one for each node would have limited the size of the problem that can be solved to lower than even the present 30; if there were one matrix per node, some execution time that is now spent regenerating matrices from the initial cost matrix would have been saved, possibly leading to slightly better speedup.

4.4. Experiments

The results presented here are for a problem size of 30 cities, the largest problem that could be solved on Cm*. Beyond size 30, the number of nodes in the state space tree required to store the state of the problem exceeded the system capacity. Inter-city distances were generated by a random number generator with uniform distribution in a convenient interval [Pascal 81, Knuth 69].

With implementation 1, these problems were solved with degrees of parallelization ranging from 2 to 16. Parallelization degree of 2 corresponds to one master and two slaves working together deciding on one edge at each iteration. (The master process is blocked when the slaves are active; so we do not count the master for degree of parallelization.) This computation, with parallelization degree of 2, forms the base for the performance comparisons for this implementation. Parallelization degree of 16 corresponds to one master and 16 slaves working on the 16 different include/ exclude selections of 4 edges in each iteration. Implementation 2 always decided on one edge at a time irrespective of parallelism; so parallelism could be any number. The experiments were conducted with 1 to 16 processes, with the single process results serving as the base for speed comparisons.

Each of the processes runs in its own dedicated processor with all its code, stack and any local data localized to that processor. This frees one from concern for the issue of localization of program objects; this aspect of parallel processing has already been well studied on Cm* [Raskin 78].

5. Results

In this section data for the two implementations on speedup, number of nodes in tree, and contention will be presented. They will be compared in section 6. Speedup is the standard measure of parallel performance. It is the ratio of serial execution time to parallel execution time and reflects the effective parallelism achieved for a given nominal parallelism. (Its definition is slightly modified for implementation 1 for practical reasons.)

5.1. Implementation one

5.1.1. Speedup

Figure 5-1 plots speedup against parallelism for a problem size of 30 for implementation 1.[1] Speedup here is computed as

2 * solution time for parallelism of two / solution time for given parallelism.

---

[1] As explained before, the degree of parallelism for this implementation can assume only values that are a power of 2. However data points in all the figures in this section (5.1) are shown interpolated to depict trends of smoothened values.

The solution time is taken to be the elapsed time for solving the problem, excluding the time spent creating the slave processes. The execution times vary depending on the distribution of processes among the clusters (this behavior will be studied separately later). To factor out this phenomenon, only the best times among all the different placements for which the experiments were performed were used.

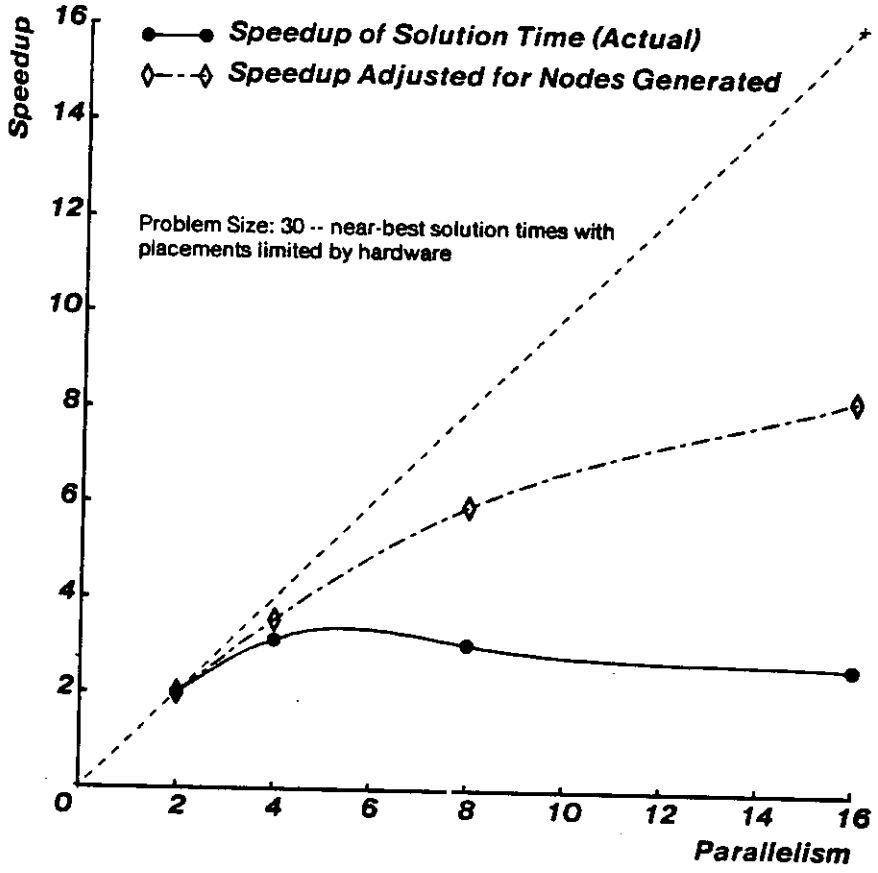

Figure 5-1: Speedup versus Parallelism for Implementation 1

The dashed line shows linear speedup -- speedup equals parallelism -- and the solid line plots the actual speedup achieved for this implementation. The actual speedup is reasonable between parallelisms of 2 and 4 (speedup for parallelism 4 is 2.8). However it starts going down after a parallelism of about 6 and remains constant at a speedup of 2.6, after a parallelism of about 8. This is explained by the greater total amount of work done with greater parallelization for this implementation and by saturation of system bottlenecks with greater amount of simultaneous computation in the system.

The dot-dashed line in the graph is an attempt to factor out the effect of the greater amount of work done with greater parallelism. This would have been the speedup of this implementation, if the total computation for solving the problem did not increase with increasing parallelism. The primary work of the algorithm consists of setting up and reducing the cost matrix corresponding to the nodes of the state space tree. So the number of nodes generated is a measure of the total computation performed for solving the TSP. Assuming that total computation for solution was directly proportional to the number of nodes generated,

speedup adjusted for nodes generated corresponding to parallelism N
= 2 * solution time per unit computation with 2 processes /
solution time per unit computation with N processes

= 2 * (solution time with 2 processes / number of nodes generated with 2 processes) /
(solution time with N processes / number of nodes generated with N processes)

The curve corresponding to the adjusted speedup behaves much better; it is a lot nearer to the linear speedup line, does not show any signs of peaking or saturation and continues to increase reasonably till a parallelism of 16, which was the maximum parallelism of the experiment. The difference between the linear speedup line and this curve corresponds to loss owing to the synchronous control of the program and to contention for system resources such as the KMaps, various system busses, SLocals, and the Object Manager.

5.1.2. Number of nodes

Figure 5-2 plots the number of nodes generated against degree of parallelism for a problem of size 30. As discussed before, this is an indicator of total amount of work done. This increases linearly with parallelism for our implementation, increasing our execution times at greater parallelisms, effectively nullifying any real speedup beyond a parallelism of about 6. In this implementation, with increasing parallelism, greater amount of computation is done between consecutive applications of heuristic bounding. This decreases the effectiveness of the heuristic and increases the search and the amount of computation required to solve the problem.

The other curve shows the number of nodes expanded, which corresponds to the number of iterations needed to solve the problem. This number is an indicator of actual execution times, when the sequential components of execution times and contention for system resources are ignored. It decreases initially but levels off at a parallelism of 6.

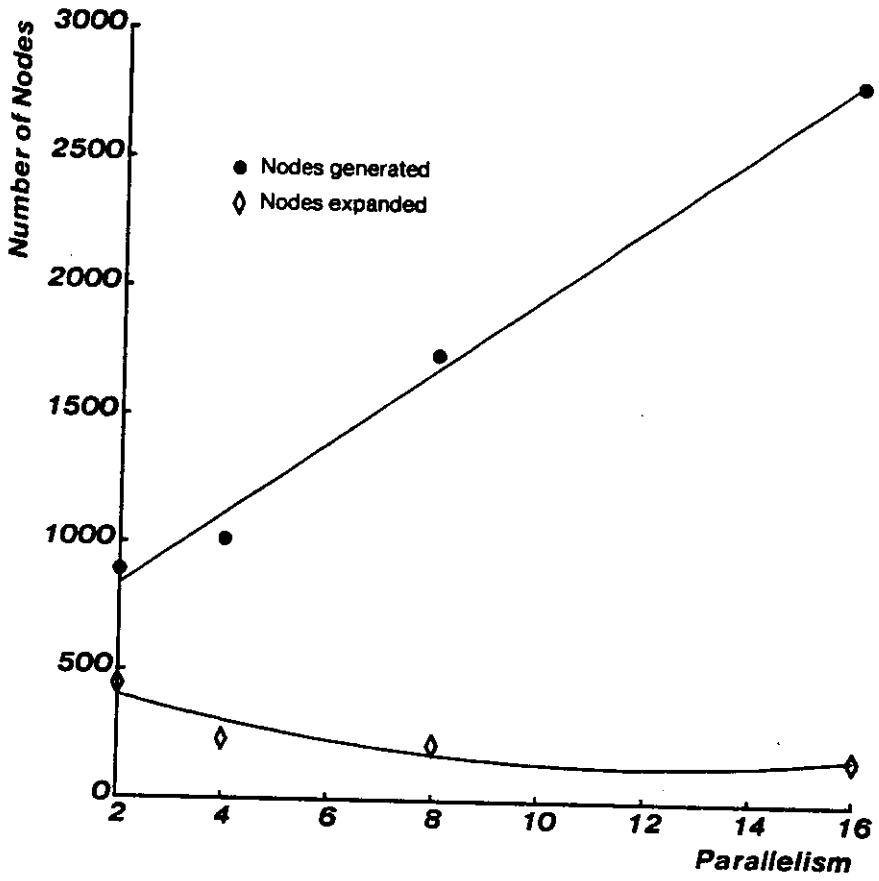

Figure 5-2: Nodes versus Parallelism for Implementation 1

5.1.3. Placement

As was mentioned before, for a given parallelism the execution times vary depending on how the processes are distributed among the clusters of Cm*. Two opposing factors affect this variation: as the number of processes in a cluster increases, there is more contention for cluster resources and this increased contention increases execution times; but at the same time, since fewer clusters are used, common data generated by the processes (the nodes of the tree) get less distributed among clusters and this decreased distribution of common data tends to decrease average access times to global data and hence the execution times.

In general, in many past multiprocessing experiments by others, execution times increase as more clusters are used, suggesting that remote access times dominate in those experiments. However, for this program, cluster contention dominates causing the execution times to decrease as the processes are distributed among more clusters.

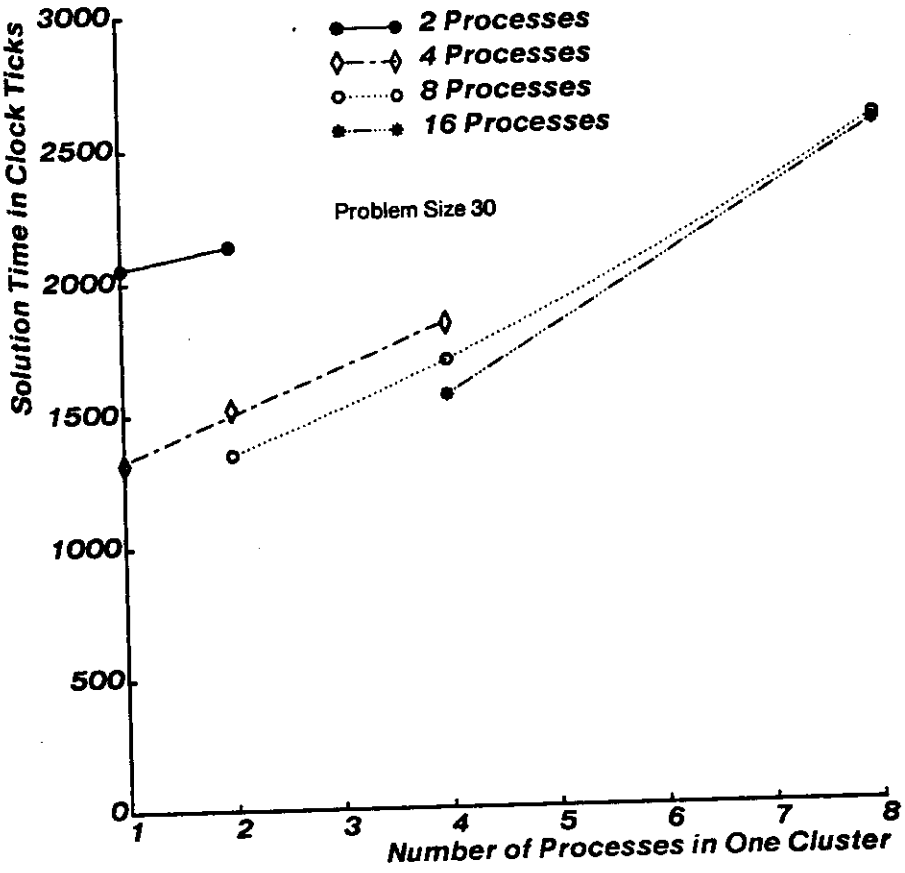

Figure 5-3: Solution Time versus Placement for Implementation 1

Figure 5-3 plots solution time against placement of processes among clusters, for a problem size of 30. The number of processes in a single cluster is plotted along the X-axis. For a given curve corresponding to a particular parallelism, cluster contention increases with increasing X, resulting in increased execution time. The execution time increases fairly linearly with the number of processes in a cluster, at about the same rate for all parallelisms.

5.2. Implementation two

5.2.1. Speedup

Figure 5-4 plots speedup against parallelism for a problem size of 30 for implementation 2. Since for this implementation single process execution is possible, actual speedup is computed as

solution time for parallelism of one / solution time for given parallelism.

Again the solution time is taken to be the best elapsed time for solving the problem under different

distributions of the processes among the clusters, excluding any time spent in creating processes, just as for implementation 1.

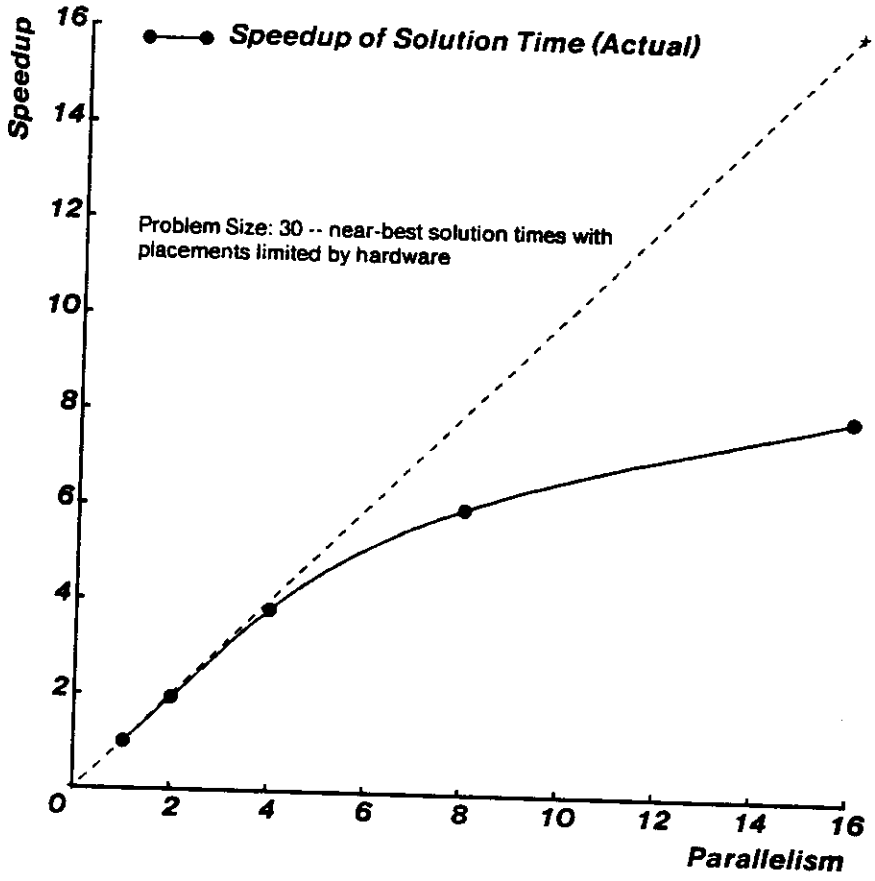

Figure 5-4: Speedup versus Parallelism for Implementation 2

Here are a few observations on the results:

- The speedup curve looks much better than in implementation 1. It is close to the linear speedup line until we reach a parallelism of 6 and does not show any signs of peaking or saturating at higher parallelisms. It is almost identical to the speedup adjusted for nodes-generated in implementation 1. This adjusted speedup was what the system was capable of achieving, if the total computation for solving the problem had remained constant with increasing parallelism. This near-identity of the two curves suggests that the amount of computation does not increase with parallelism here. The number of nodes generated remains nearly constant for all parallelisms, confirming this inference.

- The absolute execution times are lower than in implementation 1 by about 25% at a parallelism of 2 to about 80% at a parallelism of 16. For a parallelism of 2, the nodes generated and nodes expanded for both implementations are the same, suggesting that the total amount of computation done by all processes for solving the problem is about

the same for both the implementations at this parallelism. So one has to look elsewhere for an explanation of the 25% decrease in absolute execution time at this parallelism. The following are possible explanations for this behavior.

- In the previous implementation (with its synchronous program control), some time is wasted because of a lack of absolute work balance between the slave processes. In this implementation with its asynchronous control, absolute work balance between processes is not critical and there is no time lost because of this.

- Because of the egalitarian control structure of this program, processes here do not idle waiting for more work. In implementation 1 (with its master-slave structure), when the master is busy deciding on which node to expand next, the slaves are idle except for creating a look-ahead node by calling the Object Manager.

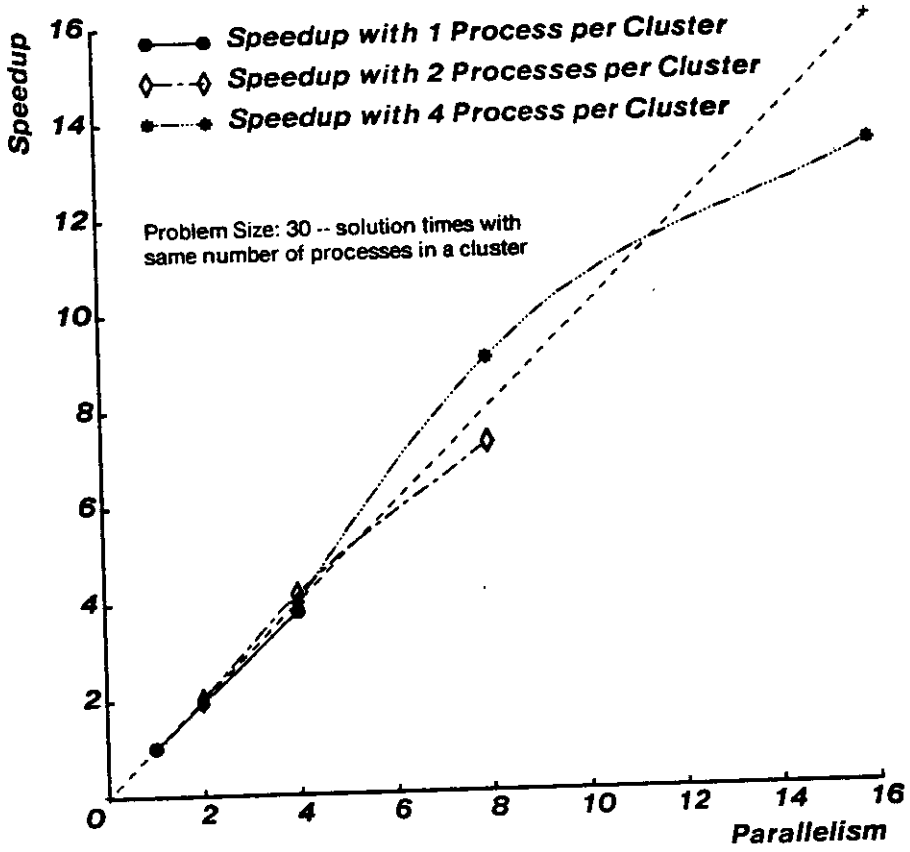

Figure 5-5: Speedup (normalized for cluster contention) versus Parallelism for Implementation 2

Figure 5-5 is an attempt to factor out cluster contention effects on speedup. Here speedup is computed using solution times with the same number of processes in a cluster. This ensures that cluster contention is near about the same for all data points on a given curve. The curves hug the

linear speedup line closely except near a parallelism of 16 suggesting that below a parallelism of 12 there is hardly any system level contention and most of the loss in execution time seen in the previous speedup figure is attributable to cluster level contention. Near a parallelism of 16, system level contention for the two locks in the program (one lock for controlling access to the ordered list of leaf nodes and another for global data) and for system resources such as the intercluster bus affects performance adversely to a more significant extent.

5.2.2. Number of nodes

With varying parallelism, number of nodes generated remained constant near 900 and the number of nodes expanded remained constant near 450. (These numbers are the same as for implementation 1 at a parallelism of 2.) This is in sharp contrast to implementation 1 where the number of nodes generated increased considerably with increasing parallelism. This single factor contributed the most to the improved shape of the speedup curve in implementation 2. The constant total computation may be explained by the constant granularity (to be discussed later) of this implementation.

5.2.3. Placement

Figure 5-6 plots solution time against placement of processes among clusters for a problem size of 30. The number of processes in a single cluster is plotted along the X-axis.

Execution time increases, for a given parallelism, as more processes are placed in the same cluster, just as in implementation 1, suggesting cluster contention dominates over global access here too. For a parallelism of 2, there is a greater absolute increase in execution time when both processes are placed in the same cluster compared to the corresponding increase for implementation 1. This may be because of the greater average parallel activity in implementation 2, especially for a parallelism of 2. The absolute increase in execution times is marginally less for parallelisms of 4 and 8 and dramatically less for a parallelism of 16 compared to corresponding figures for implementation 1. For a parallelism of 16, this increase is about 250 clock ticks for implementation 2 and 1000 ticks for implementation 1. Asynchronous execution randomizes (distributes in time) cluster resource usage and hence reduces contention.

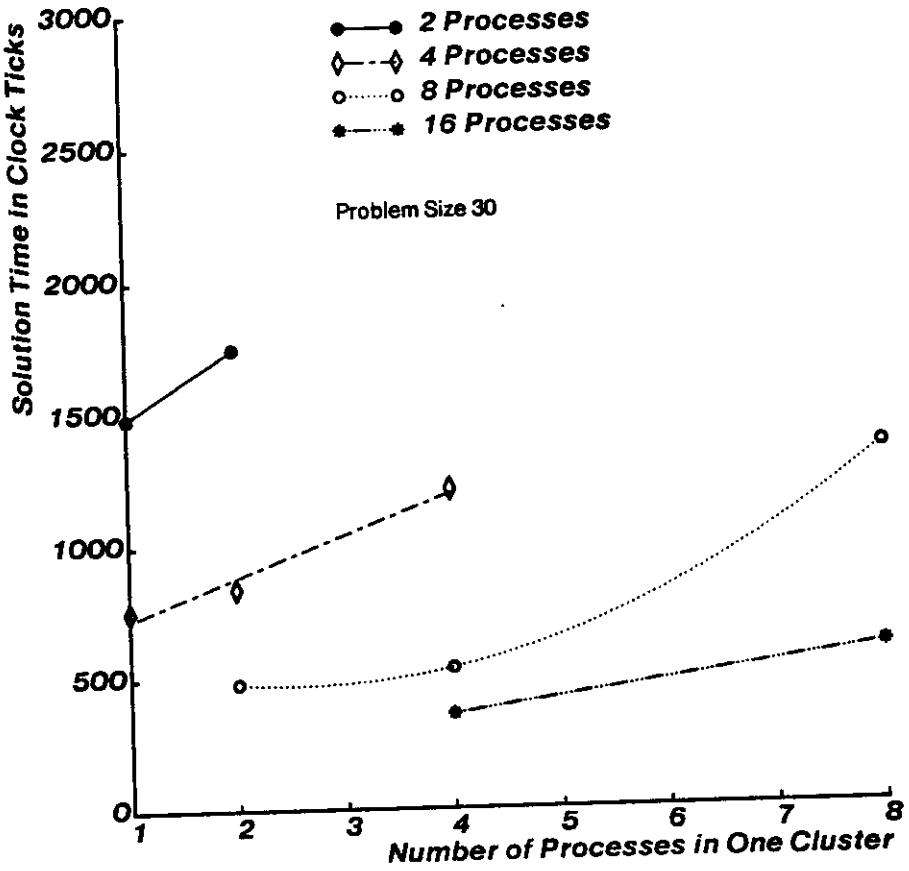

Figure 5-6: Solution Time versus Placement for Implementation 2

6. Discussion

6.1. Control synchronism

One reason for the better performance of implementation 2 is its loose asynchronous control as compared to the tightly synchronized control of implementation 1. In implementation 1, the slave processes work in a lock stepped fashion under the control of the master process. This mode of control depends heavily on the work balance between the slaves, in order for the whole computation not to be pulled down by a laggard process. The slaves are fairly, but not exactly, balanced in their workload; but still there will be some loss in execution times because of residual imbalance.

Asynchronous control made possible an egalitarian process structure, which allows all processes to do useful work all the time, except when they find a resource they need being used by some other process. In implementation 1, each tour-partitioning decision is made by the master process and during this time all the slaves are idle.

Further, since all processes in a loosely controlled execution proceed at their own pace, they use resources (such as the Object Manager) more distributed (instead of clustered together) in time, leading to reduced contention. The curves on the effect of placement of processes (figures 5-3 and 5-6) suggest that at higher degrees of parallelisms the effect of contention is less severe for implementation 2.

The price for looser control in implementation 2 is locked access to common data; in implementation 1, there are absolutely no locks since the master strictly controls the work of each slave. Apparently, at least for this algorithm, the other benefits of asynchronous control heavily outweigh the cost of locked access and the possible contention for the locks.

A simple comparison of the number of points of synchronization between the two implementations is instructive. In implementation 1, every process waits once after setting up a new child node; so the number of synchronization points equals the number of nodes generated, which increases with increasing parallelism. At each synchronization point, all slave processes wait till the master process finds the next node to expand, reduces it if necessary, and selects the edges to use in the next expansion. So there is considerable synchronization cost even with the least number of processes and this increases with increasing parallelism. In implementation 2, on the other hand, every process uses two mutual-exclusion locks each time they set up two child nodes; so the number of synchronization points here equals the number of nodes generated, which remains constant with parallelism. The processes wait only when they find another process holding the lock they need; because of asynchronous control, the requests for locks occur distributed in time, so the number of points at which a delay occurs at these synchronization points will be low, at least when the number of processes competing is low. So the total loss in execution time owing to synchronization must be low for implementation 2.

6.2. Resource contention

While the speedup characteristic of implementation 2 is reasonably good, it is far from ideal especially at higher parallelisms. As the speedup curves adjusted for cluster contention show, most of the loss in efficiency is attributable to contention for cluster resources. Cluster contention arises out of the underlying hardware and software architectures. Since Kmaps play a central role in providing the StarOS virtual machine, considerable use of them is inevitable. Also a large number of small objects (the nodes of the state space tree) are created by the programs using the StarOS utility, the Object Manager, which is a cluster-unique subsystem. While it is possible to reduce their use by deliberately trying, we desisted from doing so because we wanted to concentrate on broader issues

that are nearer to the logic of the program rather than on issues near the architectural level.

System-level contention for implementation 2 starts to play a more significant role after a parallelism of about 12. This contention must be for the locks in the program and for the intercluster bus. It may be possible to reduce the contention for the locks by using a locking scheme more complex than the present spin lock with a small wait loop. The contention for the intercluster bus arises from accesses to data lying in remote clusters. Algorithms based on tree search make considerable use of common data and seem particularly susceptible to system-level contention.

6.3. Process granularity

Process granularity here means nearly the same as what Kung calls module granularity [Kung 80]. It is the average amount of computation done by all processes between two consecutive points (instants) of communication among them. In the synchronous implementation 1, processes communicate with each other implicitly through the master process; so here the communication points are the points of synchronization. In implementation 2, processes communicate with each other indirectly through shared information in the state space tree; so here communication points are those when some process accesses the state space tree to update it or to choose the next node to expand.

How do the granularities of the two implementations compare? For both implementations the amount of computation for setting up a node at the same depth in the tree is about the same; let its average value be W units. In implementation 1, for a parallelism of N, between two communication (synchronization) points, each process computes W units and so the N processes together compute $N*W$ units. While in implementation 1 the interval between communication points remains constant with parallelism, in implementation 2 it decreases. In implementation 2, let the interval between two points when a particular process accesses the state space tree be T. As the parallelism increases from 1 to N, the average interval between two communication points (points when *any* process accesses the tree) decreases from T to T/N, taking the computation by one process between communication points from $2*W$ to $2*W/N$. This keeps the total computation by N processes constant at $2*W$. So the granularities of the two implementations are about the same for a parallelism of 2; however, as parallelism increases, the granularity of implementation 2 remains constant, while that of implementation 1 increases.

Communication exacts a cost -- an explicit cost for using some communication mechanism and an implicit cost owing to possible idleness of processes and to contention. Since, by definition, larger

granularity means more computation done between communication points, for the same total computation it also means fewer communication points leading to possibly reduced overhead for parallel execution. However in algorithms of the heuristic search variety, such as the LMSK algorithm, communication points correspond to points when some heuristic is applied to guide the search for the solution; so a larger granularity means less effective application of the heuristic and more search and work for solving the problem. The experimental data on the number of nodes generated for the two implementations support this expectation. The granularity and the number of nodes generated for implementation 1 increase with parallelism, while, for implementation 2, they remain fairly constant. For implementation 1, not only the savings from the fewer communication points but even additional computing power at greater parallelisms was completely nullified by the increase in work and the speedup is constant after some parallelism.

6.4. Parallelization

The better performance of implementation 2 is directly attributable to its better granularity characteristics, its asynchronous control, and its egalitarian process structure. These characteristics of implementation 2 were made possible by the parallelization scheme was adopted. Parallelization schemes differ in the degree of freedom that they allow a designer when making other implementation decisions for the algorithm. In implementation 1, for example, the tight control is a result of parallelizing at the level of setting up the multiple child nodes and there is no direct way to achieve loose control; implementation 2, however, lends itself easily to asynchronous control (as well as tight control, if we so choose), with each process expanding a node without being synchronized with the others. The parallelization scheme that one decides to adopt will have to be one that will make possible a wider spectrum of implementation decisions; it should permit the choice of those program characteristics that lead to better performance.

6.5. Parallel search

The paradigm of multiple processes asynchronously setting up multiple nodes of a state space tree through which they share global information -- the paradigm of implementation 2 -- seems to be a general parallel problem solving scheme applicable to most, if not all, tree-searching algorithms.

On a preliminary examination of the suitability of different kinds of algorithms for adaptation to parallel execution, heuristic search algorithms (of which branch and bound algorithms are a special case) seemed less amenable for parallelization at a gross level than other kinds of algorithms, such as segmentation or divide and conquer. This fear arose from two general properties of such search algorithms. They make extensive use of global data without any predictable access pattern

throughout the computation, leading to a large number of expensive remote accesses and increased contention. Experimental data bears out this expectation. The other property is the generally serial nature of heuristics. Heuristics guide computation toward the solution incrementally as the computation progresses towards the solution. We surmised that when the computation is done in parallel, the heuristics become less effective in searching for the solution, leading to more total computation to solve the problem. As it turned out, however, the serial nature of the heuristic search does not seem to affect total computation significantly in implementation 2. This is in spite of LMSK algorithm proving to be a fairly focussed one. The above pessimistic argument has no cognizance of the concept of granularity, which is the key to the understanding of this behavior.

Acknowledgements

I thank Anita Jones, Zary Segall, and Gerald Thompson for their guidance during this work and Robert Whiteside for his discussions.

References

[Bellmore 68] Bellmore M. and Nemhauser G. L.
The Traveling Salesman Problem.
Operations Research 16, 1968.

[Christofides 79] Christofides N.
The Travelling Salesman Problem.
In Christofides N., Miagozzi, Topf and Saudi (editor), *Combinatorial Optimization*, pages 131-149. Wiley, 1979.

[Hoffman 81] Hoffman A.J., Johnson E. L., Wolfe P., Held M.
Aspects of the Traveling Salesman Problem.
"Research Report" RC 8787, IBM, April, 1981.

[Horowitz 78] Horowitz E. and Sahni S.
Fundamentals of Computer Algorithms.
Computer Science Press, 1978.

[Jones and Gehringer 80]
Anita K. Jones and Edward F. Gehringer [eds.].
The Cm multiprocessor project: A research review.*
Technical Report CMU-CS-80-131, Computer Science Department, Carnegie-Mellon University, July, 1980.

[Karp 62] Karp R.M. and Held M.
A Dynamic Programming Approach to Sequencing Prolems.
J. Soc. Indust. Appl. Math. 10(1), Mar, 1962.

[Karp 70] Karp R. M. and Held M.
The Traveling-Salesman Problem and Minimum Spanning Trees.
Operations Research 18, 1970.

[Karp 71] Karp R. M. and Held M.
The Traveling-Salesman Problem and Minimum Spanning Trees: Part II.
Mathematical Programming 1, 1971.

[Knuth 69] Knuth D. E.
The Art of Computer Programming: Seminumerical Algorithms.
Addison-Wesley, 1969.

[Kung 80] Kung H. T.
Advances in Computers. Volume 19: *The Structure of Parallel Algorithms.*
Academic Press, 1980, .

[Lawler 66] Lawler E. L. and Wood D. E.
Branch-and-Bound Methods: a Survey.
Operations Research 14, 1966.

[Levin *et al.* 76] Roy Levin, David Jefferson, and Joseph Newcomer [eds.].
C.mmp Linker Reference Manual.
Technical Report, Computer Science Department, Carnegie-Mellon University, August 19, 1976.

[Little 63] Little J. D. C., Murty K. G., Sweeney D. W. and Karel C.
An Algorithm for the Traveling Salesman Problem.
Operations Research 11, 1963.

[Mitten 70] Mitten L. G.
Branch-and-Bound Methods: General Formulation and Properties.
Operations Research 18(1):24, 1970.

[Nilsson 71] Nilsson N. J.
Problem-solving Methods in Artificial Intelligence.
McGraw-Hill, 1971.

[Pascal 81] Hisgen A.
PASCAL on the DEC System 10 at CMU
Carnegie-Mellon Univ., 1981.

[Raskin 78] Levy Raskin.
Performance evaluation of multiple processor systems.
PhD thesis, Carnegie-Mellon University, August, 1978.
Published as technical report CMU-CS-78-141.

[StarOS 81] Gehringer E.F., Chansler R. J.
StarOS User and System Structure Manual
cmucsd, 1981.

[Swan 78] Richard J. Swan.
The switching structure and addressing architecture of an extensible multiprocessor, Cm.*
PhD thesis, Carnegie-Mellon University, August, 1978.

[Wulf *et al.* 70] W. Wulf, *et al.*.
BLISS-11 Programmer's Manual.
Computer Science Department, Carnegie-Mellon University, 1970.