# The Traveling Salesman Problem:

## The Development of a Distributed Computation

*Nick Lai*
*Barton P. Miller*

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720

## ABSTRACT

The Traveling Salesman Problem (TSP) is computationally expensive to evaluate. It can, however, be readily decomposed into subproblems that can be computed in parallel. Developing a distributed program taking advantage of such a decomposition, however, remains a difficult problem.

We developed such a distributed program to compute the TSP solutions, using a new set of distributed program performance tools to better understand our TSP program. These tools allowed us to discover the performance bottlenecks in our program and to revise the program to significantly improve its execution speed.

This research was supported by National Science Foundation Grant MCS-8010686, the State of California MICRO program, and the Defense Advance Research Projects Agency (DoD) Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies either expressed or implied, of the Defense Advance Research Projects Agency or of the United States Government.

## 1. Introduction

When a programmer implements a program that uses parallel computation, the goal is typically to increase the speed of the computation. Unfortunately, this goal is difficult to obtain. One reason for this is that communication between processes can be slow. This cost becomes greater when processes on different machines communicate. To obtain the goal of increased speed, the programmer must develop algorithms that minimize the effect of this communication cost.

It is essential for the programmer to be able to measure the performance of his or her program. The programmer needs to debug the program, and to know where the program is spending its time so that it can be modified or redesigned to increase its execution speed. The problems of debugging and performance monitoring become more difficult in a distributed environment due to a general lack of tools to aid the programmer. Tools for measuring distributed program performance have been implemented for Berkeley UNIX [Miller, Sechrest & Macrander 84]. These tools allowed the development of the distributed program described here.

To examine the process of program development, we chose a problem that lends itself to this examination, the Traveling Salesman Problem. This problem is computationally intensive, but has well known solutions. We developed a program to find solutions to the problem without using parallel computation and, using that program as a basis, developed programs that would solve the problem using parallel processing. We debugged and measured our programs' execution using tools designed to passively measure parallel performance [Miller, Sechrest & Macrander 84]. Using the information gained from these measurements, we were able to improve our programs and to measure how much these improvements enhanced performance.

## 2. The Traveling Salesman Problem

### 2.1. Description of the Problem

The Traveling Salesman Problem (TSP) is a popular problem among both operations research experts and computer scientists. In general, the problem can be stated as follows: given N cities and the costs associated with going from each city to each of the other cities, find a minimum cost circuit that visits each of the N cities once and only once. Formally we have a *problem graph* G = (V, E) where V is the set of vertices (cities) and E is the set of edges, each representing the cost of going from each city to another. In a problem of size N, we have N vertices and m = N(N - 1) edges. The problem can also be represented as an N x N matrix, with each element $c_{ij}$ being the cost of going from city $i$ to city $j$. This matrix is called the *representative matrix* of the problem.

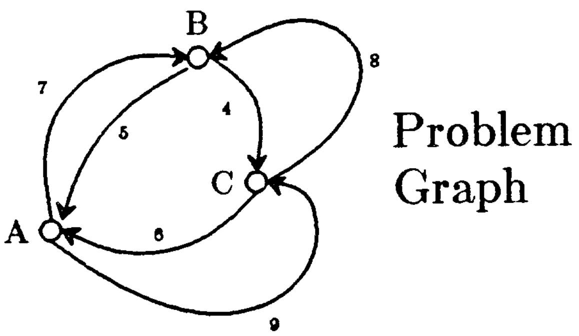
We address the problem in its most general form. In particular, we address the case of the asymmetric non-euclidean TSP. The problem is *asymmetric* because, in general, $c_{ij}$ does not equal $c_{ji}$. The problem is *non-euclidean* because the costs are arbitrary nonnegative numbers, unconstrained by the triangle inequality. There are many possible algorithms that can be used to efficiently solve the TSP. We have decided to follow the algorithm presented in [Mohan 82].

### 2.2. General Algorithm

We use a *branch-and-bound* algorithm to solve the TSP. This algorithm is based upon the development of a *state tree* in which we record the paths that the TSP program has examined and the costs associated with those paths. Each node in the state tree corresponds to a collection of edges in the problem graph that make up a *partial path* through the problem. Each node of the state tree also contains a lower bound cost. The lower bound cost is the least cost that can be obtained by including that partial path in a full circuit. It is calculated by taking the representative matrix, and for each edge $c_{ij}$ setting all elements in row $i$ and column $j$ of the matrix to 0, and then reducing the matrix. The constants used to reduce the matrix are then added to the summation of the cost of each edge in the partial path; the result is the lower bound cost.

By examing the lower bound costs associated with the nodes of the state tree, the program can intelligently choose the best path to examine. This allows us a potentially large reduction in computation time with respect to that needed by solutions which compute the cost of every possible circuit through the problem. A decision to examine a particular path is called a *branching* decision, and it is based upon an examination of the lower bounds of all of the leaf nodes of the state tree [Horowitz & Sahni 78]. Every time a branching decision is made, new nodes are created in the state tree that correspond to the possible extensions of the selected partial path.

Figure 2.2.1 shows an example of how a TSP problem is translated from a problem graph to a representative matrix to a state tree. In this example we have a small (three vertices, six edges) problem. The first representation is the problem graph, which shows the vertices and the directed edges of the problem. The representative matrix shows the cost of going from each of the three vertices to each of the other reachable nodes. Finally, the figure shows the development of the state tree. Vertex $A$ of the problem graph is arbitrarily selected as the root node of the state tree, and, by reducing the representative matrix, we find that the problem's lower bound is 17. We next set up the state tree nodes that correspond to the possible edges that include $A$, namely, $AB$ and $AC$. After setting row A and column B in the representative matrix to 0, we reduce the matrix with constants that sum to 10. Adding these constants to the cost of edge $AB$, which is 7, we find that the lower bound for a solution containing the edge $AB$ is 17. Similarly we find that the lower bound for $AC$ is 22. Examining all of the leaf nodes of the state tree, we decide to pursue the possibilities of a path containing $AB$, and set up nodes in the state tree for all of the problem graph edges that may be used after including $AB$ in a solution. There is only one such problem graph edge, $BC$, and a state tree node is set up for this edge, and its lower bound is computed to be 17. Once again, we examine the lower bound costs of the leaf nodes of the state tree, $BC$ and $AC$, and choose $BC$ because it has the lowest lower bound. Only one more edge exists beyond $BC$: $CA$, which completes the circuit and solves the problem with a cost of 17, which is lower than any other possible circuit through the problem.



To

| From | A | B | C |
|---|---|---|---|
| A | ∞ | 7 | 9 |
| B | 5 | ∞ | 4 |
| C | 6 | 8 | ∞ |

Representative Matrix

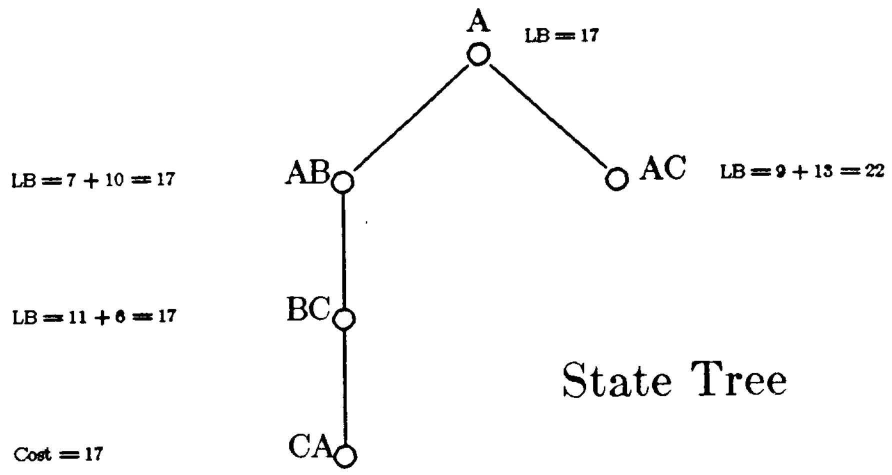


**Figure 2.2.1: Representations of the TSP**

## 3. Computational Environment

### 3.1. VAXs and 4.2BSD UNIX

The program was developed on a Digital Equipment Corporation VAX 11/750 with 2 megabytes of memory. Occasionally, a second VAX of the same configuration was used for data collection. These VAX computers are connected by a 3 megabit Ethernet.

The operating system running on these VAXs is Berkeley UNIX 4.2BSD. This version of Berkeley UNIX supports virtual memory and processes with address spaces up to 16 megabytes. Most importantly, it supports interprocess communication [Leffler, Joy & Fabry 84].

### 3.2. Interprocess Communication

We used the interprocess communication facilities built into 4.2BSD UNIX. The primary object of communication under this facility is the *socket*. A socket is an endpoint of communication. These sockets can be connected to form bidirectional communication streams.

For our purposes, it was most effective to test our program on a single machine and then use the gathered measurements to predict performance on multiple machines. The *socketpair()* system call of Berkeley UNIX [Joy et al. 83], which returns an unnamed pair of connected sockets, was used to create a pair of connected sockets; internally, these are represented as file descriptors.

A current restriction of Berkeley UNIX is that it allows a given process to have a maximum of 20 open file descriptors. Our program consists of a master process that submits computation requests to a number of server processes. The master process must have a socket in common with each of its server processes. Reserving one file descriptor for error messages, we are therefore limited to having 19 server processes.

## 4. Measurement Tools

This section briefly describes the tools used to measure and analyze the performance of our program. These tools are more fully described in [Miller, Sechrest & Macrander 84] and [Miller 84].

### 4.1. Data Collection

There are three basic components to the data collection process. First, special functions in the kernel collect data about communication events. In particular, for a message sent or received, information collected includes the sender of the message, the receiver of the message, the message length, the CPU time associated with the event, and other related data. Second, the kernel hands this information to a *filter* program, that selects the event records that the user plans to analyze later and saves these traces in a file in a user-specified format. The third major component is the user interface to these measurement tools, provided by a control program that coordinates the actions between the various parts of the measurement system. The control, filter and user processes may run on the same or on different processors.

### 4.2. Data Analysis

The raw data produced by the measurement tools is analyzed by a program that uses them to build a table showing the interactions between all of the processes. It summarizes the number of packets sent and received by each process, the sizes of those packets, and the processor time consumed by each process. The program generates a report on the amount of parallelism achieved by the program, given various assumptions about transmission costs. It can also project performance in the case of multiple CPU's for various assignments of processes to processors.

## 5. Evolution of a Program

The unit of measurement of performance that we use in this paper is the parallelism factor, **P**. This is a measure of speed-up in the computation, or how much of our computation is done in parallel. In the case where there is no parallel execution, **P** is 1; in the case where there are $n$ processes in a computation and each process is concurrently performing $\frac{1}{n}$ of the work, **P** is equal to $n$. For this study, improving the performance of the computation means, for a given problem size (number of cities), increasing the value of **P**. In section 5.6 we comment on the effect that speed-up has on machine utilization.

In the measurements of the TSP programs, we use two forms of the factor **P**. The first form of **P** assumes that communication delays between processes are zero (i.e., communication is instantaneous), and that no process competes for the CPU with any other process. This gives an upper bound on the amount of parallelism that could be achieved by the measured execution of the program. The second form of **P** incorporates communication delays and contention caused by multiple processes executing on the same machine. The parallelism analysis can project the value of **P** for different assignments of processes to machines. A complete description of the derivation of the parallelism factor can be found in [Miller 84].

During the course of this research project, we developed three primary versions of the TSP program. These are referred to (in order) as the Non-Blocking Server, the Low-Node Caching and the Overlapping Requests versions. Note that each of these versions represents successive improvements to its predecessors, and *not* complete re-implementations.

Due to time considerations, we were not able to measure the performance of all of our implementations over a large variety of TSP problem sizes. For comparison purposes, all the implementations were measured for problems of size sixteen, with four, eight, twelve and sixteen processes.

## 5.1. Single Process Solution

We began by writing a single process implementation of our algorithm. This allowed us to become familiar with the TSP while ignoring the complexities of interprocess communication.

The basic algorithm of the program is as follows: First a matrix representing the problem is created. Second, the root node of the state tree is created. Each node of the state tree is identified with an edge's entry from the representative matrix, and represents a partial path through that matrix. This partial path can be obtained by traversing up the state tree to the root node. At each step of this traversal, the edge that the state tree node represents is a part of the partial path. The node also holds the lower bound cost associated with the partial path of which it is a part, and the lower bound cost of the solution that could be obtained by following that partial path. A node of the problem is arbitrarily selected as the starting point of the solution.

In each iteration of the main loop of the program, a node is selected as having the best possible chance of being on the optimal path. This is done by traversing the entire state tree and choosing the leaf node with the lowest lower bound value. This node represents the path most likely to yield an optimal circuit.

Child nodes are now set up for the chosen node. For each node that could be reached from the chosen node (that is, all of the reachable cities that are not already in the path), a child node is created, and lower bound costs for the child node are computed.

This loop is repeated until a complete path is found. The first complete path found will be an optimal path because the cost associated with it will be less than or equal to the cost associated with any other path.

## 5.2. Multiple Process Solution

There are many possible approaches to converting this computation into a parallel computation. We introduce parallelism by creating *child* processes that do some of the calculations needed by the *parent* process for each iteration. In other words, each child performs the part of the calculation that it is given, and all of the decisions are made by the parent process.

Most of the computation time is spent when new nodes are added to the state tree and the lower bound cost for each new node must be calculated. Our implementation of this program calculates these lower bound costs for each new node in parallel. When the program starts, a number of child processes are created. Instead of having the single parent calculate the lower bounds, these computations are handed to the child processes in the form of requests. Later, the results are returned to the parent, which stores the values in the appropriate node.

In the first multiple process version, the parent sends requests to each of the children and then awaits responses from all of the children before sending out the next batch of requests.

Although this version of the program ran and produced accurate answers, we did not start the actual performance measurements on our program until we had solved the simple problem described in the next section.

## 5.3. Non-blocking Server

After examining the program and its performance, it became apparent that most of the benefits of parallel execution were being lost because the parent process waited until all children sent back answers before sending out the next batch of computation requests. This was solved by having the parent hand out new requests to a child as soon as it returned an answer. We considered this version to be our first successful implementation, and the following measurements show its performance.

Figures 5.3.1, 5.3.2 and 5.3.3 show the projection of how this implementation of the program performed. Figure 5.3.1 shows the achievable degree of parallelism for problems with various matrix sizes and numbers of processes. This graph assumes that communication costs are zero, and that the processes do not compete with each other for the CPU. Figure 5.3.2 presents the performance of the program when each process is run on its own machine, over varying problem sizes and numbers of processes. Figure 5.3.3 illustrates the amount of parallelism achieved by our program when two processes are allocated to each processor under a variety of problem sizes and number of processes.

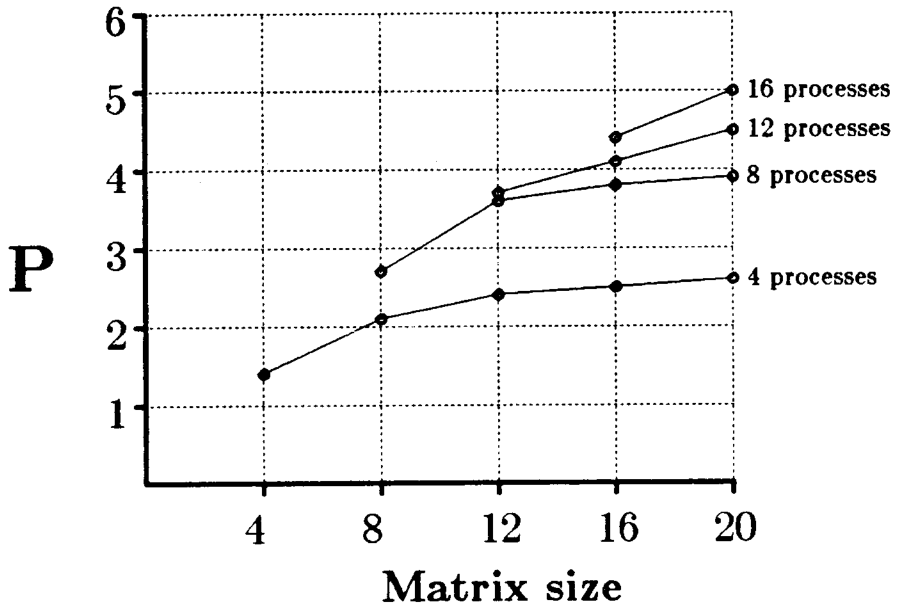


**Figure 5.3.1:**
**Upper Bound Parallelism of the Non-Blocking TSP Program.**

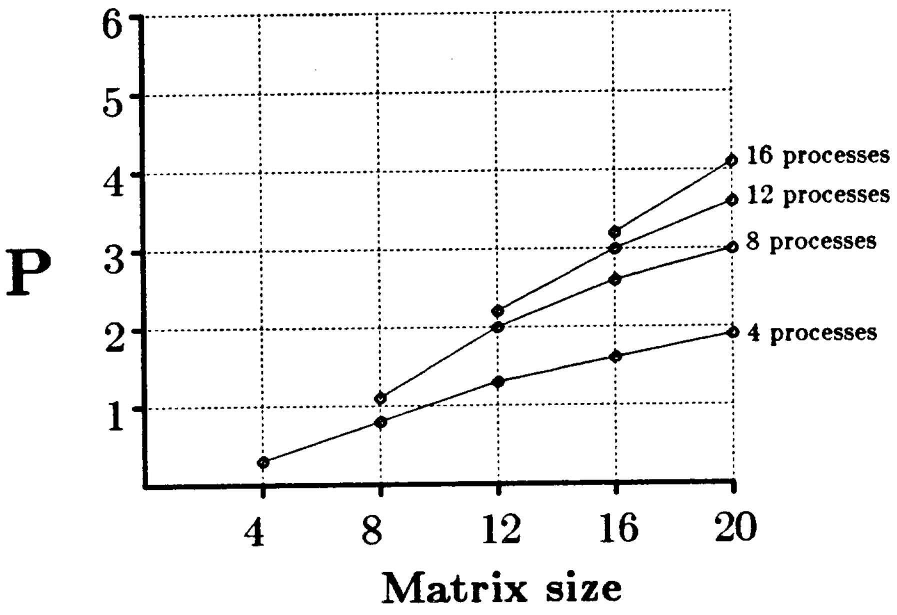


**Figure 5.3.2:**
**Parallelism vs. Matrix Size, One Process per Machine.**

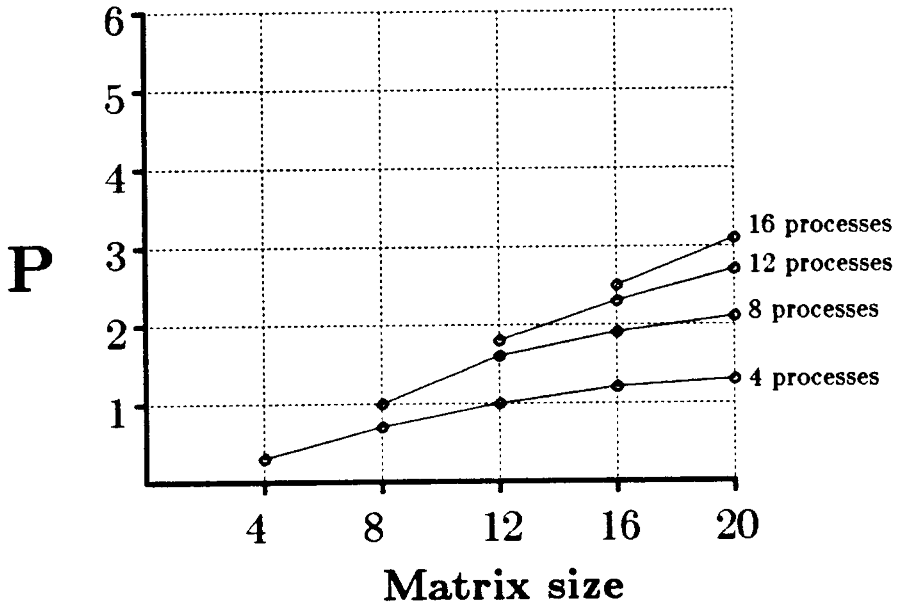


**Figure 5.3.3:**
**Parallelism vs. Matrix Size, Two Processes per Machine.**

The results show that the amount of parallelism achieved increases with the size of the problem. As the problem size increases, a larger percentage of the program's time is spent computing the lower bounds. Since we do the computation of the lower bounds in parallel, it is natural that the degree of parallelism should increase with the problem size.

If we compare the corresponding curves from graphs 5.3.1 and 5.3.2 for a single problem size, we can see the cost (loss of parallelism) from communications delays. Comparing graphs 5.3.2 and 5.3.3, we see the loss of parallelism when two processes contend for a single CPU.

## 5.4. Low-Node Caching

The process status of the running program showed that it was using large amounts of memory. This memory was obviously being used for the state tree. However, this meant that we were spending a large percent of program time searching through the state tree for the lowest node. During this time the child processes were sitting idle, and parallelism was being lost.

To eliminate this waiting time, we decided to save 100 of the previously calculated nodes with the lowest values in an array. For each child node that is created, if the array is not full or if the value of the child node is less than the cost of the last (highest value) element in the array, this value is inserted, in order, into the array. The result is an ordered array of the leaf nodes from the state tree with the smallest values. Our branching decision is made by selecting the leaf node of the state tree that has the smallest value associated with it, in this case the first node in the array. When this node is selected, it is eliminated from the array.

Performance measurement of this Low-Node Caching version of our program shows a significant improvement in the parallelism achieved by the program. Figure 5.4.1 shows the results of the measurements of this version. Our measurements where made for problems of size 16 with 4, 8, 12 and 16 processes. Figure 5.4.2 compares this Low-Node Caching version with the Non-Blocking Server version.

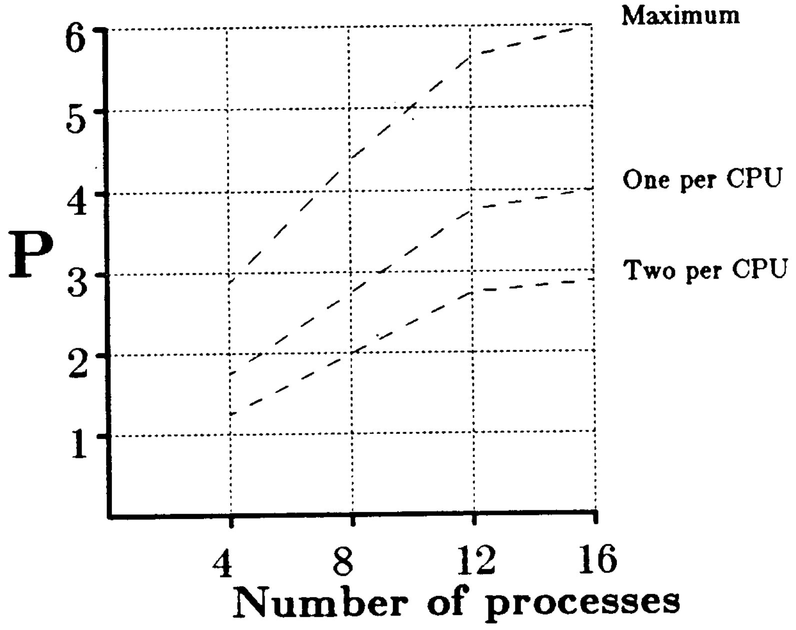


**Figure 5.4.1:**
**Parallelism vs. Matrix Size, Low-Node Caching Version**

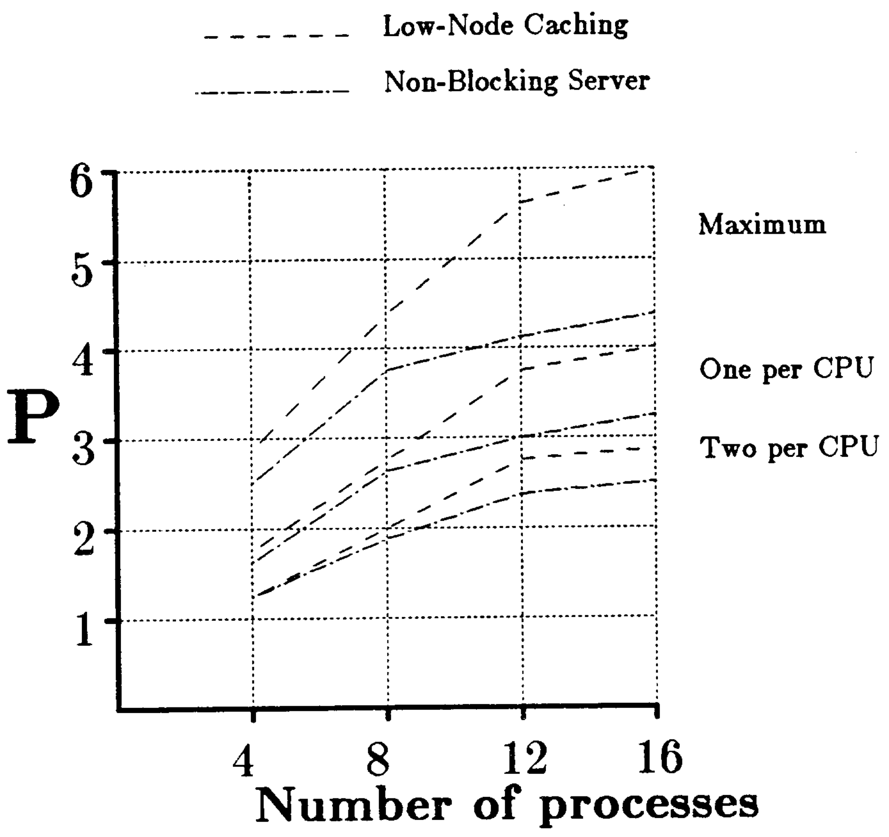


**Figure 5.4.2:**
**Low-Node Caching versus Non-Blocking Server**

The above figures show that this Low-Node Caching version performed significantly better than the non-blocking server implementation. We also find that this improvement in performance is more pronounced as the number of processes increases.

This improvement is the result of the parent process doing less computation relative to the computations it gives to its children processes. That is, less time is spent in the parent process in this version, while the children are still doing the same amount of computation as in the Non-Blocking Server version.

## 5.5. Overlapping Requests

Experimental measurements show that communications costs are actually very high, taking approximately 8 ms for communications between processes on a single machine and 20 ms for processes on different machines [Hunter 84]. This means that a child process remains idle while information is transferred back to its parent and while waiting for a new request to arrive from the parent.

In order to eliminate the idle time in the case where the number of requests exceeds the number of servers, we overlapped the requests to the children by initially sending out two requests to each child. Children no longer had to wait for the parent to receive the result before beginning work on a new request, since another request was usually waiting. This is an important innovation, and is aimed solely at trying to cut down the effect of the communications cost. As the following results show, this minor change enhanced the program's performance.

Figure 5.5.1 presents the parallelism achieved by the Overlapping Requests implementation for a problem of size 16 with 4, 8, 12 and 16 processes. Figure 5.5.2 shows how the Overlapping Requests version compares to the Low-Node Caching version. Figure 5.5.3 shows the performance of the Overlapping Requests implementation, as well the performance of the previous two versions, for a problem of size 16 with 4, 8, 12 and 16 processes.

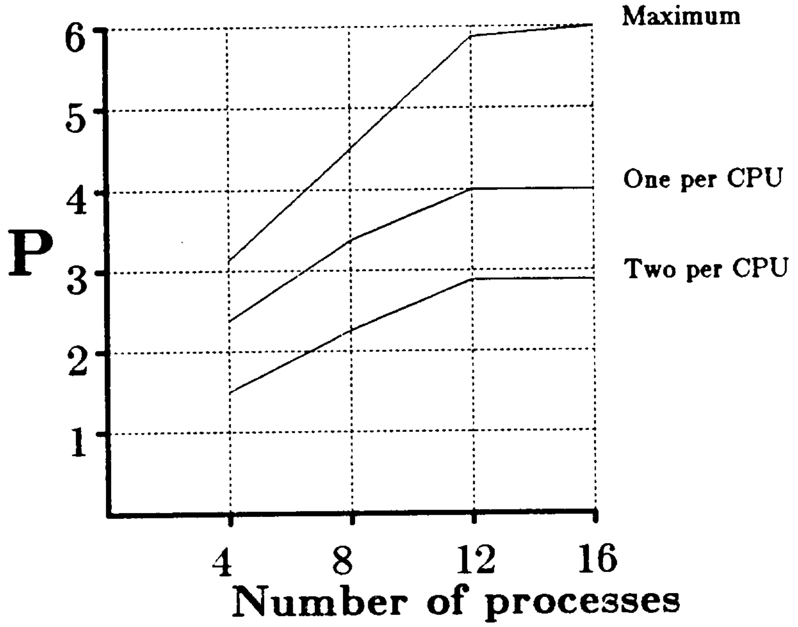


**Figure 5.5.1:**
**Parallel performance of the Overlapping Requests Implementation.**

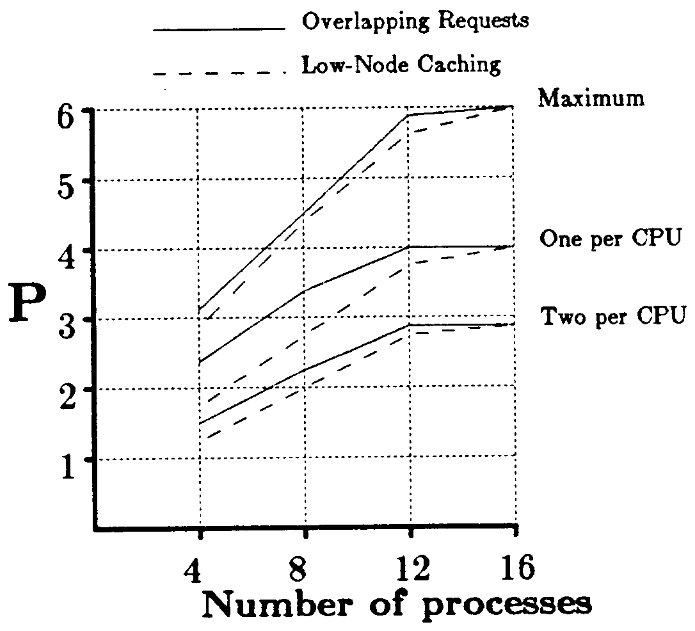


**Figure 5.5.2:**
**Overlapping Requests versus Low-Node Caching.**

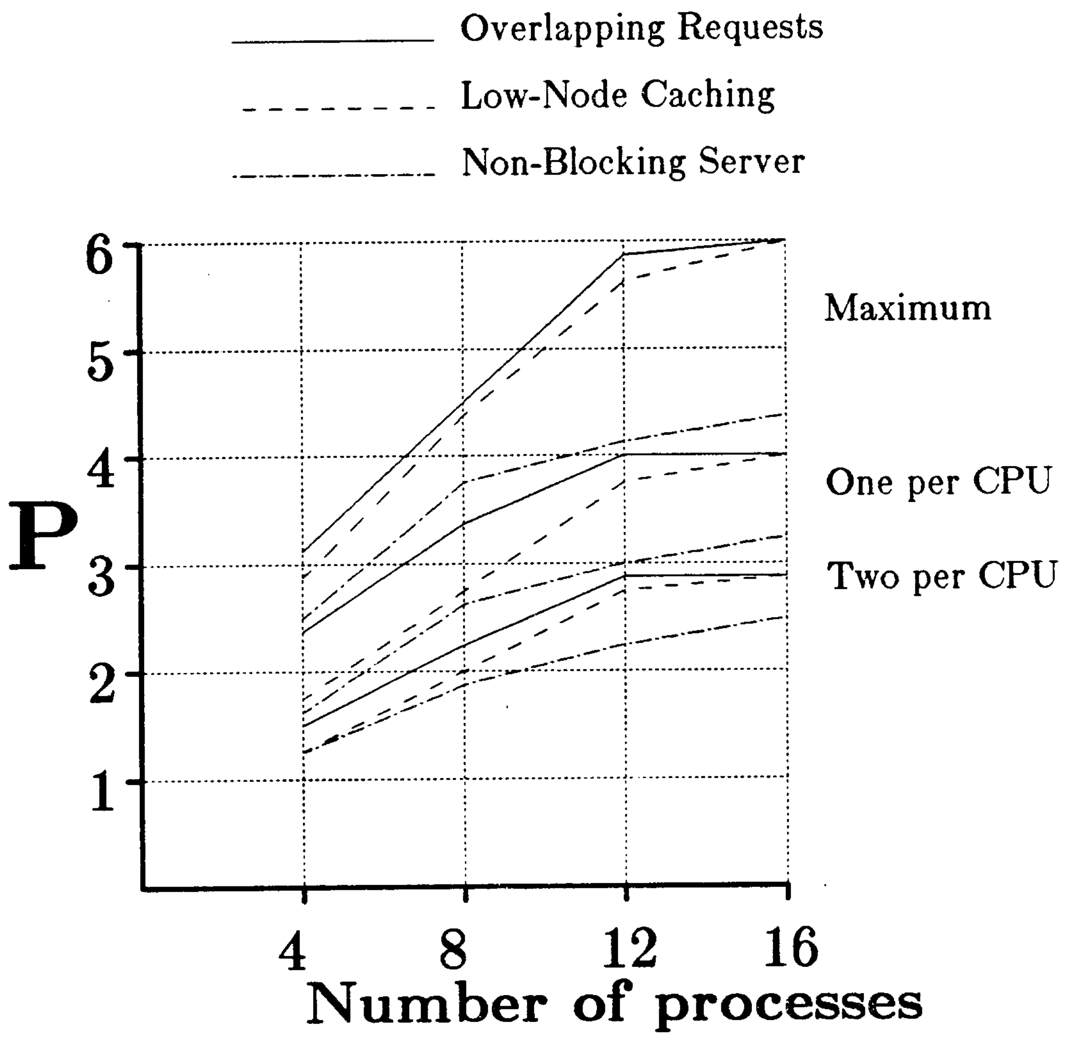


**Figure 5.5.3:**
**Comparison of the Three Implementations.**

Note that the Overlapping Requests version performs better than the Low-Node Caching version when the number of processes is small, but that the difference in performance between the two diminishes as the number of processes approaches the size of the problem. If there are as many processes as the size of the problem, then all of the computations are done by having each child do only one computation, thus there is no opportunity to use overlapping to reduce communication delay effects.

### 5.5.1.1. Is Faster Always Better?

In this study, we have based our evaluation of program performance on the amount of parallelism or speed-up achieved in the different versions of the TSP program. By using **P** as our only evaluation criterion, we have stated: faster is better. This view of performance dictates that even a small increase of parallelism is worth the addition of more hardware.

It is also important to know how well we are using our computing resources. This can be stated as: how much of our available computing resources are we using? For the TSP programs, we ask: how much of the machines (CPU's) involved in the computation are we utilizing? We can compute CPU utilization from the parallelism factor. We define utilization as

$$\text{Utilization} = \frac{\mathbf{P}}{N}$$

where $N$ is number of machines used for in the computation. Figure 5.6.1 graphs the values of the utilization (corresponding to the graphs in Figure 5.5.3) for the three versions of the TSP program.

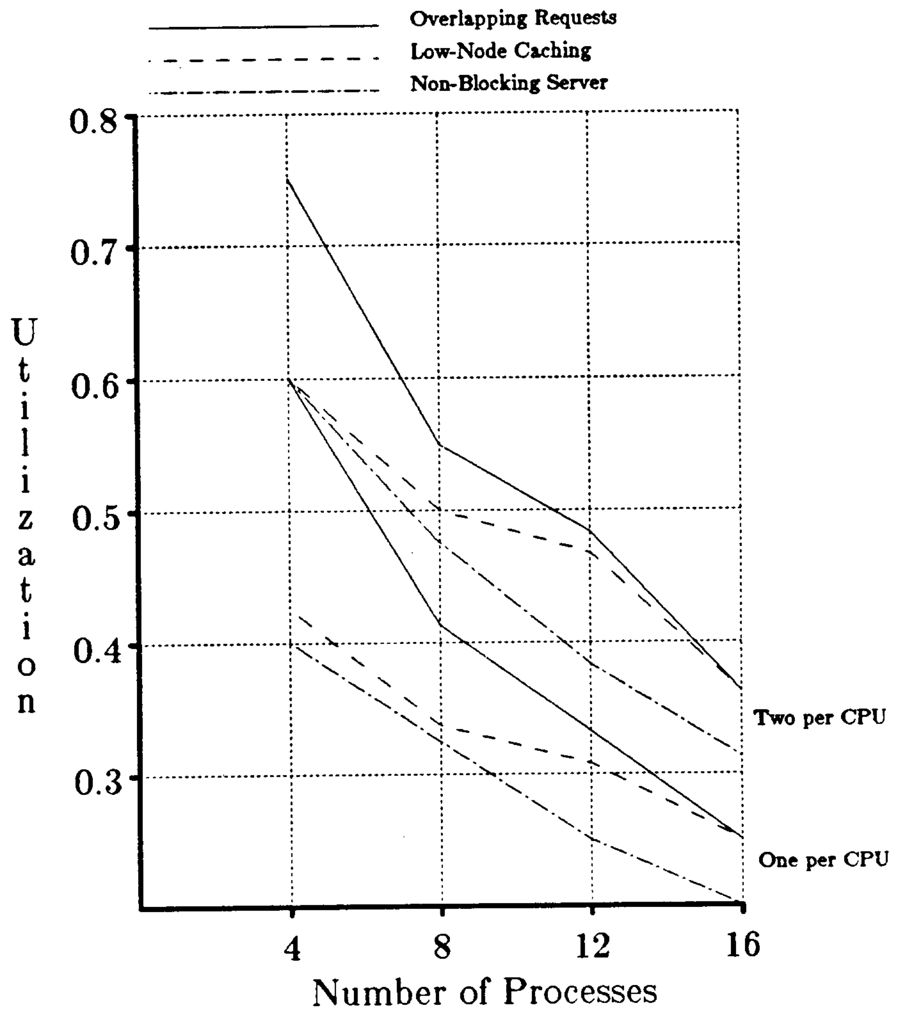


**Figure 5.6.1:**
**Machine Utilizations for the Three Implementations**

Our first observation from the graph in Figure 5.6.1 is that when two processes run on each machine, we have a higher utilization (and from Figure 5.5.3, a lesser amount of parallelism) than when one process runs on each machine. This means that while running one process per machine is a less efficient use of resources, it results in a faster execution.

Our second observation from Figure 5.6.1 is that each successive version of the TSP program made better (more efficient) use of the CPU's. We know that the successive versions are better (and not just requiring more computation time) since the amount of CPU time need for each successive version also decreased. The fact that the curves for the Overlapping Requests version and the Low-Node Caching version meet when 16 processes are used reflects the fact that the problem size is the same as the number of processes.

A person who uses a distributed program, such as the TSP program, wants his/her program to execute as quickly as possible. The parallelism factor, **P**, gives a measure of how much we can increase the speed of execution. It is also important to know how efficient is a program. We need metrics such as CPU utilization evaluate efficiency.

## 6. Conclusion

The purpose of this study was to evaluate a collection of measurement tools for distributed programs through the development of a simple program. The question was: could these tools help us better understand our program, and help us improve its performance?

While the main intent of this study was to evaluate the performance of a distributed computation, the measurement tools also proved useful for debugging. When the first version of the TSP program was being tested, its execution speed seemed slower than expected. A visual examination of the trace records produced by the measurement tools quickly (that is, in less than one minute) showed that the master process in the computation was only creating a single child process. This was caused by a simple typing error in the program. After this error was corrected, the performance seemed unchanged. A second inspection of the traces showed that, while the correct number of child processes were being created, the master was sending requests only to a single child - again a typing error. After this, the program worked correctly. The discovery of these errors is an indication that the measurement tools are providing some of the additional information needed for developing distributed programs.

The main direction of this study was the investigation of the performance of the TSP program. In particular, we wanted to maximize the amount of parallel activity in the program. The distributed program measurement tools, in conjunction with other existing tools, provided the information necessary to measure the program's performance, identify bottlenecks, correct the problem, and evaluate the corrections.

Traditional performance tools provide information as to *how much* time a program spends in its various procedures and modules. Our measurement tools in conjunction with the analysis of parallelism also provide an insight into *when* a program spends its time in these modules.

## References

[Hunter 84]
E. Hunter, "A performance study of the Ethernet under Berkeley UNIX 4.2BSD", Technical Report UCB/CSD, University of California, Berkeley, 1984.

[Leffler, Joy & Fabry 84]
S. Leffler, W.N. Joy, and R. Fabry, "4.2BSD Networking Implementation Notes," Technical Report UCB/CSRG, University of California, Berkeley, July 1983.

[Joy et al. 83]
W.N. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, and D. Mosher, "4.2BSD System Manual," University of California, Berkeley, July 1983.

[Miller 84]
B. Miller, "Performance Characterization of Distributed Programs", Thesis, University of California, Berkeley, 1984.

[Miller, Sechrest & Macrander 84]
B. Miller, S. Sechrest, and K. Macrander, "A distributed program monitor for Berkeley UNIX", Technical Report UCB/CSD, University of California, Berkeley, 1984.

[Mohan 82]
J. Mohan, "A study in Parallel Computation – The Traveling Salesman Problem," Technical Report CMU-CS-82-136, Carnegie-Mellon University, August 1982.