Parallel Dynamic Tables

by

Daricha Techopitayakul

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Engineering in Electrical Engineering and Computer Science

and

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1995

Author ..... Daricha Techopitayakul .....
Department of Electrical Engineering and Computer Science
May 26, 1995

Certified by .....
Charles E. Leiserson
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Thesis

Parallel Dynamic Tables

by

Daricha Techopitayakul

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 1995, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Electrical Engineering and Computer Science
and
Bachelor of Science in Computer Science and Engineering

Abstract

This thesis explores issues involving construction and application of a dynamic size table on a parallel system, one that grows and shrinks as available memory size changes. Motivation behind this project lies in a new concept of variable-size memory resource. Traditionally, memory allocated for a program computation is always fixed at the beginning of runtime. For computation on a parallel or distributed system, however, the number of available processors and their memory may be dynamic. Processors may join and leave the computation, as they become idle or later occupied. This project studies how to utilize all obtainable memory at any instance. It experiments with a method of integrating available memory dynamically to a data structure table, so the table varies its size depending on the memory level. It also examines the related parameters of expanding and contracting the table that will maximize the use of memory at any moment while still preserving a feasible cost of the table. The result shows that with the Phish system's pattern of usage, a high fraction of available processors of up to 95% – 96% should be utilized because the aggregate number of available processors normally varies within a limited range. Therefore, even though the cost of reshuffling table entries is high, reshuffles rarely happen. Same analysis can be applied to other systems, given their patterns of processor usage. Moreover, the project suggests an alternative of adjusting the fraction of processor utilization dynamically during the runtime. The result shows that this algorithm also yields an optimal performance, while no knowledge of the processor usage pattern or the calculation of optimal parameters are needed.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

Parallel Dynamic Tables

by

Daricha Techopitayakul

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 1995, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Electrical Engineering and Computer Science
and
Bachelor of Science in Computer Science and Engineering

Abstract

This thesis explores issues involving construction and application of a dynamic size table on a parallel system, one that grows and shrinks as available memory size changes. Motivation behind this project lies in a new concept of variable-size memory resource. Traditionally, memory allocated for a program computation is always fixed at the beginning of runtime. For computation on a parallel or distributed system, however, the number of available processors and their memory may be dynamic. Processors may join and leave the computation, as they become idle or later occupied. This project studies how to utilize all obtainable memory at any instance. It experiments with a method of integrating available memory dynamically to a data structure table, so the table varies its size depending on the memory level. It also examines the related parameters of expanding and contracting the table that will maximize the use of memory at any moment while still preserving a feasible cost of the table. The result shows that with the Phish system's pattern of usage, a high fraction of available processors of up to 95% – 96% should be utilized because the aggregate number of available processors normally varies within a limited range. Therefore, even though the cost of reshuffling table entries is high, reshuffles rarely happen. Same analysis can be applied to other systems, given their patterns of processor usage. Moreover, the project suggests an alternative of adjusting the fraction of processor utilization dynamically during the runtime. The result shows that this algorithm also yields an optimal performance, while no knowledge of the processor usage pattern or the calculation of optimal parameters are needed.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

Acknowledgments

I would like to take this opportunity to thank Professor Leiserson for guiding me through the thesis. His devotion to work and his personality are inspiring. Thanks for his technical advice and his words of wisdom throughout the year. I am also grateful to Bradley for helping me debug my code and giving me advice, even at a very low level. I always wonder if I will ever know that much when I finish my Ph.D. Thanks to Yuli, Keith, Christ, Bobby, Rob, Howard and Richard for creating such a friendly and supportive environment to work in.

My deepest gratitude goes to my parents for their understanding and encouragement. If it hadn't been for them, I wouldn't have come this far. Thanks for believing in me and for giving me the strength. I also want to thank both of my sisters for their moral support and sharing throughout my difficult times.

Thanks to Jimmy, Paulus, Phoebe, Thomas and Jocelyn for starting off our MIT lives together. MIT is more enjoyable with their friendship. I also want to thank William for always helping me out when I needed the most. His understanding is beyond what friends across cultures could imagine.

I also want to thank my Thai friends at MIT for making MIT more like home. My special thanks to Nong Meaw for her care and encouragement. Also, thanks to my friends at CMU for their friendship and for making CMU my second home in the US. Last but not least, I want to thank my best friend, Pek, for his technical and emotional support, and for walking by my side throughout these years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

This research studies an organization and the feasibility of a dynamic table on a parallel system. An example of an applicable environment is the Phish system, a network of workstations where processors come and leave [2]. When processors become idle, i.e. when users do not log on to the terminal, we want to incorporate those processors and their memory into our computation. At the same time, when some processors want to leave, we need to adjust the table management so that our computation works correctly after the processor is released.

A dynamic parallel table can be applied to the computation that employs a large, global table. The correctness of this computation, however, must not depend on the persistency of information within the table. The reason is that when the table contracts, it may discard some information that the table no longer has space to store. The characteristics of storing non-vital but useful information resembles that of cache. The larger the cache size is, the faster the computation can run. The dynamic size table serves this purpose, so it expands to the largest size possible, given the memory available.

In order to solidify the experiment, I implemented a variable-size transposition table for the ⋆Socrates chess program [4] on Cilk, a multithreaded runtime system [1]. The ⋆Socrates chess program employs the Cilk runtime system to search the next

move in a chess game concurrently. It then uses a global table called *a transposition table* to store information of its extensive tree searches. Analogous to a cache, the existence of entries in the transposition table is not critical to the correctness of the chess program. Nonetheless, the overall computation of the chess program speeds up if entries are available [5]. Because maintaining the dynamicity of the table incurs extra costs of bookkeeping, copying and reshuffling entry information when processors are added and removed, this thesis analyzes the table cost and the associated parameters corresponding to the real applications of the ⋆Socrates chess program and the Phish system.

1.2 Dynamic Table versus Static Table

A dynamic table allows an application program to expand its data structure, i.e. a table, momentarily when more memory becomes available. Capturing this extra utilization can increase the efficiency of the application program. To understand the advantage of a dynamic table over a static one, consider the chess program application on the Phish system.

To construct a fixed-size global transposition table on the Phish system, a minimum number of processors must be available throughout the computation. Out of 40 to 50 workstations in the Phish system in a typical ten-day period, only five processors are available at all time[1]. Using the performance model for the chess application from chapter 4, the dynamic size table performs 19.65% better than a static table, as shown in the following calculation.

Let the performance model for a chess application be

$$\Delta work = \Delta t \times size^{0.125} \ .$$

A static table would accomplish the work of $847,395 \times 5^{0.125} \approx 1.04 \times 10^6$, while the optimal work accomplished by a dynamic table is 1.24×10^6. [2] Therefore, a dynamic

[1] The Phish system pattern is given in figure 4-2.
[2] The result is referred from chapter 4.

table out-performs a static table by almost 20%.

In general, a dynamic table becomes more useful when the number of available processors fluctuates vastly. Thus, the amount of processors guaranteed to be available at all time is unpredictable or minimal. The static table, therefore, utilizes only a small portion of the total memory.

Chapter 2

Avoid thrashing

A simple strategy to construct a dynamic table is to use all available processors, but this method may lead to *thrashing*. Every time a processor is added or removed from the computation, the table adjusts its size and reshuffles all entries to their right positions. If the processor addition and removal occur alternatively, the table will not expand to a bigger size but will still suffer from the reshuffling and bookkeeping costs [6, pp. 367-374]. The thrashing behavior is generally undesirable[1]. Especially, if the reshuffling cost is high, an application program may waste computation cycles on the table activities rather than accomplishing its primary work.

In order to avoid thrashing, I introduce *a utilization fraction*, p, which has a value from 0 to 1. The dynamic table incorporates a portion p of the available processors to construct the global table. In general, the table tries to maintain its size within the range from $p\times$ (total number of available processors). This property is reflected in the criteria for expansion and contraction of the table.

When adding many processors causes a substantial deviation from a portion p of utilization, the table expands to incorporate fraction p of the number of available processors.

[1]Thrashing may be acceptable if a long delay separates the processor addition and removal. In this case, an application program may achieve enough performance gain from a bigger table before it spends another interval resizing the table.

Precisely, the **expansion** occurs when

$$N_a < p^2 \times N_t \ ,$$

and after the expansion

$$N_a = p \times N_t \ ,$$

where

N_t = the total number of available processors,

N_a = the number of processors currently used to construct the table (*active processors*).

As we aim to optimize the use of memory, the table shrinks only when there are no unused processors and a processor must be removed.

Thus before the **contraction**, we have

$$N_a = N_t \ .$$

After the contraction, we have

$$N_a = p \times N_t \ .$$

Notice that there is a hysteresis in the dynamic behavior. If p is less than 1, the table expands only after a few processor additions, or when N_t increases from N_a/p to N_a/p^2. Also, the table contracts only after a few processors are removed, or when N_t decreases from N_a/p to N_a. Thus, adding and removing a processor does not lead to table resizing every time. By introducing p and the above criteria for expansion and contraction, thrashing is avoided due to the lag between the table expansion/contraction and the processor addition/removal.

Chapter 3

Organization and Cost Measurements

3.1 Organization

A dynamic table is composed of smaller tables distributed on a number of processors. Each component table has the same fixed size[1](a constant number of entries); therefore, more processors constitute a bigger aggregate table. Two lists, namely *an unused-processor list* and *a mapping table*, keep track of which processors are available for the computation.

An unused-processor list contains the processor numbers that are currently available, but not used in the table.

A mapping table contains information of processors that are currently used to construct the global table. It maps virtual processor numbers, the ones used by computation, to real processor numbers. Processors in the mapping table are called *active processors.*

When a processor is added or removed from the computation, the dynamic table behaves as follows:

[1]To simplify the initial design, a table on each processor has a constant size. We may generalize each component table to be a dynamic table itself, however, by varying its associativity.

1. When a processor is added, it is inserted into an unused-processor list, thus increasing N_t. If the ratio of N_a to N_t becomes too low, and the criterion for expansion is satisfied, the table grows to size $p \times N_t$, and all entries are reshuffled.

2. When a processor is removed, it is taken out of an unused-processor list if the processor is inactive[2] at the moment. If the processor is active, and there are some processors in an unused-processor list, the processor entries are copied into a spare processor before the processor is removed. If there are no spares in an unused-processor list, the table contracts, and all entries are reshuffled. As the table becomes smaller, some entries may be discarded if collisions occur.

3.2 Costs of Dynamic Table

To maintain a dynamic table, several costs are associated with adding and removing a processor. The following statistics of costs are measured using my implementation of a dynamic transposition table for the ⋆Socrates Chess program on CM5. The transposition table varies its size from 0 to 32 processors. Measurements are repeated for different sizes of a component table (a table on each processor). The following are results categorized by the type of costs.

1. **a bookkeeping cost** to update an unused-processor list and a mapping table. This cost is incurred every time a processor is added or removed. The experimental results show that a bookkeeping cost is relatively small, ranging from 50 to 80 microseconds. It is also independent of a table size and the number of active processors. This outcome can be expected because the maximum length of both lists is 32. Therefore, it should not delay the search noticeably.

2. **a copy cost**. This cost is incurred when a currently active processor (used in the table) is to be removed, and there is a spare processor on an unused-processor list. Entries are copied to a spare processor, which then becomes

[2] A processor is not *inactive* when it is not a part of the global table.

an active processor. The initial processor is subsequently released. The global table does not change the size during this process. Table 3-1 shows the copy costs for different sizes of a component table.

3. **a reshuffling cost.** A reshuffling cost is incurred when the table expands or contracts. Table 3-1 shows the reshuffling costs for different sizes of a component table. Note that the reshuffling cost increases substantially, more than double in most cases, when the number of entries in a component table doubles. The delay results from the sequential nature of the reshuffling process on the number of lines in a component table.

Table 3.1: Reshuffling costs and copy costs of the transposition table for the ⋆Socrates chess program on CM5 as a function of a table size on one processor.

TTSIZE	Reshuffling Time (sec)	Copy time (sec)
2^{15}	0.034124	0.018304
2^{16}	0.090800	0.042402
2^{17}	0.361552	0.132052
2^{18}	0.858180	0.299859
2^{19}	0.802737	0.359138
2^{20}	1.154651	1.074019

TTSIZE = a number of lines in a table on one processor

Experiments, however, show that the reshuffling cost is independent of the number of active processors. In other words, doubling the number of active processors does not observably increase the reshuffling time. Because each processor executes the reshuffling process concurrently, the total reshuffling time remains constant, provided that the network communication bandwidth is not saturated.

From the statistics, we can conclude that the reshuffling time dominates the cost of administering a dynamic table. Moreover, the reshuffling time does not change when the number of active processors increases, while it grows substantially with the number of entries on one processor. The result suggests that increasing the number of processors instead of the number of entries on a processor is more advantageous if a larger dynamic table is desired.

Chapter 4

Optimal Analysis

Tuning a parameter p for an optimal result is experimental. A small p ensures that the table does not expand or contract too frequently. Therefore, the application program, which employs a dynamic table, can spend most of its time on its primary computations. On the other hand, a larger p implies a bigger table. If the application performance depends on the size of the table, the overall efficiency may increase by trading some reshuffling time for a bigger table.

Different p's can significantly vary the application performance. For instance, the result in the subsequent sections shows that work accomplished by the chess program during 9.8 days of the Phish run can be as low as 8.5×10^5 for the worst p and as high as 1.24×10^6 for the best p. Thus, choosing a sub-optimal p can cause the performance loss of over 30%.

In order to analyze an optimal p for a system, three kinds of information are needed: the reshuffling cost of the table, the value of a bigger size table, and the pattern of processor usage of a system. The higher the reshuffling cost is, the smaller p should be, as a high p implies frequent reshuffling. The more valuable the table size is, the higher p should be, as the application gains more by having a bigger table. And lastly, if processors do not join and leave the computation frequently, a higher p is beneficial, because reshuffling seldom occurs. The table cost results and analysis are provided in the previous chapter. The following sections in this chapter will explore the latter two factors: the value of the table size and the pattern of processor usage.

4.1 Chess Performance and Transposition Table Size

Most chess programs use a transposition table to cache the results of searches on moves. Hsu argues that increasing the transposition table size by a factor of 256 can easily improve the performance by a factor of 2 to 5 [3].

If we assume that the performance of the application program is a polynomial function of the table size, the model thus implies the following equation:

$$\Delta work = \Delta t \times size^{\alpha} .$$

From Hsu's argument,

$$\begin{aligned} W_1 &= size^{\alpha} , \\ W_2 &= (256 \times size)^{\alpha} . \end{aligned}$$

If performance improves by a factor of 2, we have

$$\begin{aligned} W_2 &= 2 \times W_1 , \\ (256 \times size)^{\alpha} &= 2 \times size^{\alpha} , \\ 256 \times size &= 2^{1/\alpha} \times size , \\ 2^{1/\alpha} &= 256 , \\ \alpha &= 0.125 . \end{aligned}$$

If performance improves by a factor of 5, we have

$$\begin{aligned} W_2 &= 5 \times W_1 , \\ (256 \times size)^{\alpha} &= 5 \times size^{\alpha} , \end{aligned}$$

$$
\begin{aligned}
256 \times size &= 5^{1/\alpha} \times size\,, \\
5^{1/\alpha} &= 256\,, \\
\alpha &= 0.29\,.
\end{aligned}
$$

Thus, from Hsu's argument, doubling the size of the table increase the performance by a factor of $2^{0.125} = 1.09$ to $2^{0.29} = 1.223$. Hsu's prediction matches perfectly with Kuszmaul's [5] experimental results of StarTech serial implementation. Table 4-1 shows performance of Kuszmaul's serial implementation of StarTech as a function of transposition table size, and its performance increase factor when the table size doubles.

Table 4.1: Performance of Kuszmaul's best serial implementation of StarTech as a function of transposition table size, and its performance increase factor when the table size doubles.

Transposition Table Entries	Time (seconds)	Performance Increase Factor
2^{16}	13506.36	
2^{17}	12670.01	1.066
2^{18}	10925.46	1.160
2^{19}	9605.36	1.137
2^{20}	8040.08	1.195
2^{21}	7138.08	1.126
2^{22}	5799.31	1.231

From the agreement of Hsu's prediction and Kuszmaul's experimental results, we can conclude that, in general, chess performance increases by 9% to 22% when the table size doubles. Moreover, if we use the performance model of $\Delta work = \Delta t \times size^{\alpha}$, the choice of α for the chess application should be 0.125 to 0.29.

4.2 Analysis of p and Pattern of Processor Usage

Given the reshuffling cost, an optimal utilization fraction p also depends on the pattern of processor usage. If processors do not join and leave the computation frequently, a higher p improves the overall performance because the table is bigger, and reshuffles rarely occur. In contrast, if the the number of available processors swings up and down dramatically and frequently, it is not efficient to use a large portion of available processors, or a large p, because a considerable amount of time would be spent expanding and contracting the table.

4.2.1 Random Walk: p and the Reshuffle Rate

To gain an initial understanding of the relationship between p and a pattern of processor usage, first assume that processors come and go in the same manner as a random walk. The next occurrence is equally likely to be a processor addition or a processor removal. Figure 4-1 shows the number of reshuffles as a function of p, given a random walk model.

Given a random walk model, the number of available processors is likely to vary within a small range because successive additions without a removal or successive removals without an addition is improbable. Therefore, reshuffles rarely happen except when p is extremely high. A random walk model suggests that if a pattern of processor joining and leaving resembles a random walk, a high p is beneficial.

4.2.2 Phish System: p and the Reshuffle Rate

In order to predict how well a dynamic size table fits the Phish system, first we can analyze the pattern of processor joining and leaving of the Phish system, which is shown in figure 4-2. Notice that the total number of processors in the Phish system fluctuates around twice a day when most users login in the morning and log out at night.

Stimulating the Phish pattern, figure 4-3 shows the number of reshuffles that occur as a function of p, if a dynamic size table is implemented on the Phish system. Note

Figure 4-1: Graph displays a number of reshuffles as a function of p, given a random walk pattern of a processor addition and removal, a total of 100 times. The graph suggests that a high p is beneficial for a random walk pattern because the table does not bear the reshuffling cost until p gets extremely high.

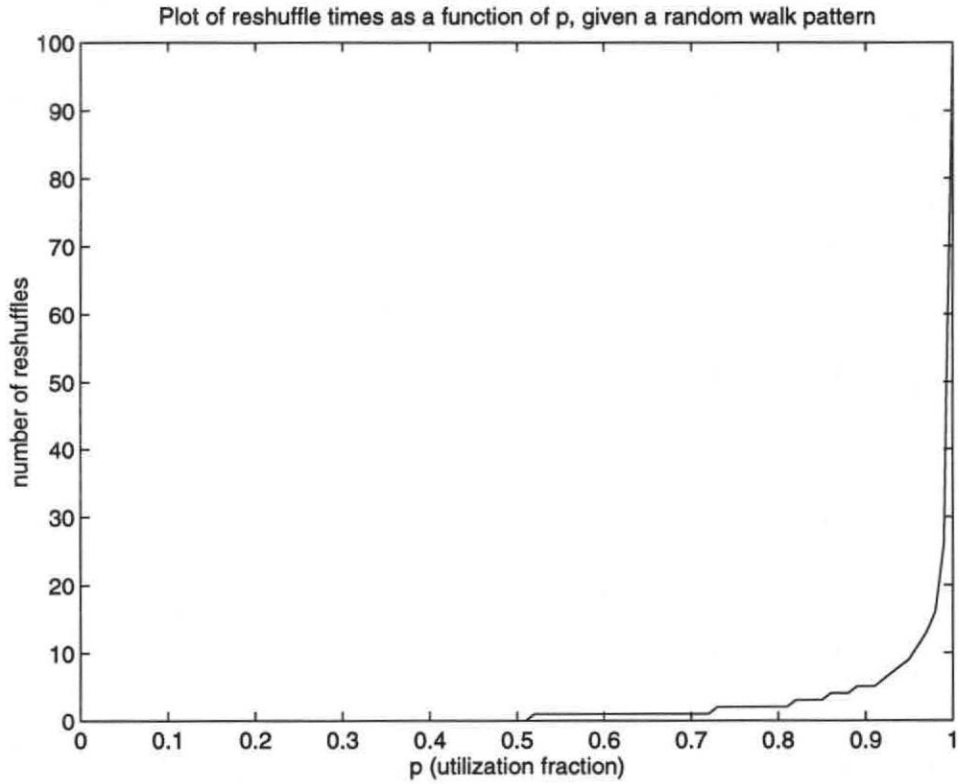
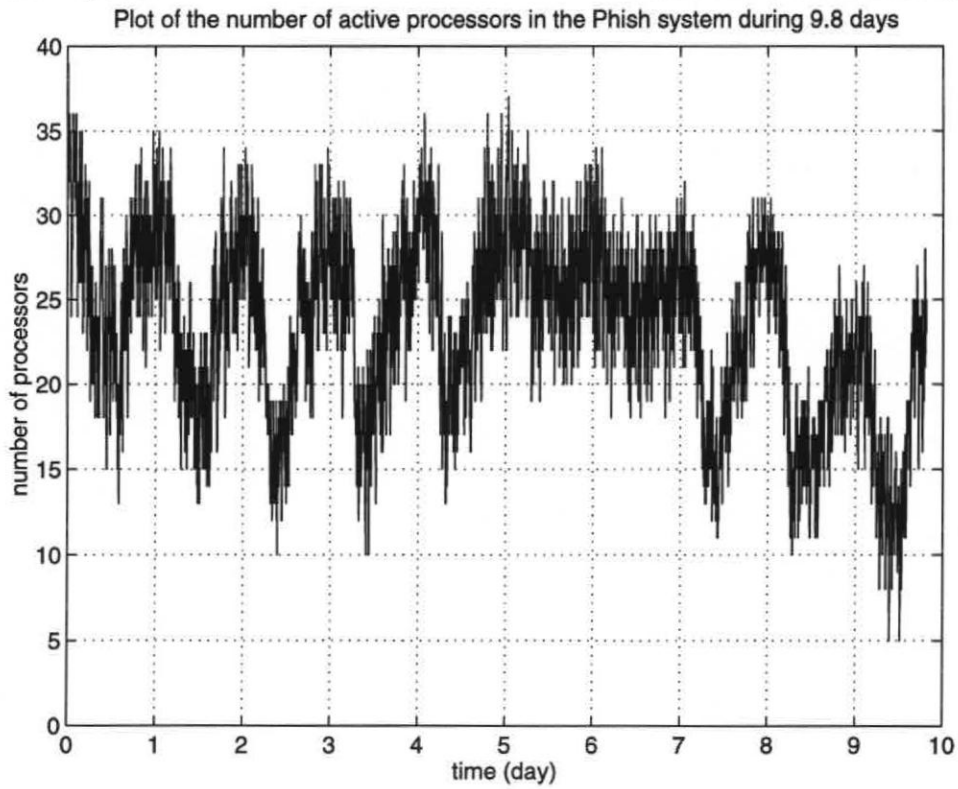
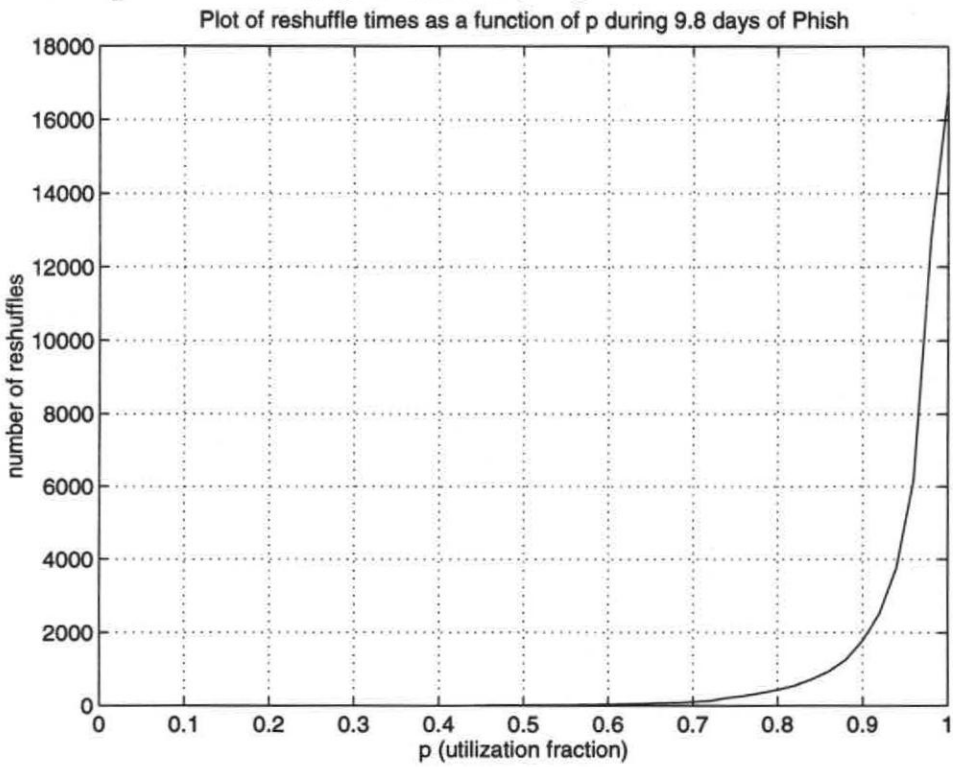


Figure 4-2: Graph displays the number of active processors in the Phish system during 9.8 days of Phish. There are a total of 45-50 machines in the Phish system.



that the pattern of reshuffle times of the Phish system is close to that of a random walk. Because the number of processors in the Phish system tends to stay around the average, except shifting only twice a day, the table can incorporate a high portion of p into constructing the table without many reshuffles.

Figure 4-3: Graph displays a number of reshuffles as a function of p, given the Phish pattern of processor usage during 9.8 days. Similar to the random walk, the graph suggests that the Phish system also benefits from a high p, as the number of reshuffles is not significant until p gets extremely high.



4.3 Analysis of Optimal p for the Phish System

Having the information about the costs of a dynamic size table, the value of a transposition table size and a system's pattern of processor joining and leaving the computation, we can calculate an optimal p for the Phish system.

Referring to table 3-1, in order to simplify the model, I will assume that the

reshuffling cost is 1 second[1], and it dominates other costs[2]. Then, consider the Phish pattern, which generates a relationship of the reshuffle times and p in figure 4-3. A derivative analysis of these statistics yields an optimal p for the Phish system.

Extracted from figure 4-3, the following data is based on 847395 seconds (or 9.8 days) of Phish run.

p	the number of reshuffles
0.89	1504
0.90	1787

An example of one step of calculation is as follows:

For $p = 0.89$, the dynamic table reshuffles every $\frac{847395}{1504} = 563.43$ seconds. Therefore, a performance efficiency, compared to when dynamic table is not present, is $\frac{563.43-1}{563.43} = 0.998225$, given that the reshuffle time is 1 second.

Applying the same calculation to $p = 0.90$, the dynamic table reshuffles every $\frac{847395}{1787} = 474.20$ seconds, therefore, has performance efficiency of $\frac{474.20-1}{474.20} = 0.99789$. Notice that efficiency decreases as p gets bigger because the table reshuffles more often.

Therefore, the performance loss of going from $p = 0.89$ to $p = 0.90$ is a factor of $\frac{0.998225}{0.99789} = 1.000335$. In short, due to more frequent reshuffling, we lose a factor of 1.000335 by raising p from 0.89 to 0.90.

From Hsu's prediction and the performance model of chess application: $\Delta work = \Delta t \times size^{\alpha}$, where $\alpha = 0.125$ to 0.29. Going from $p = 0.89$ to $p = 0.90$, the table size increases by a factor of $\frac{0.90}{0.89} = 1.011236$. Thus, performance gain from a bigger table is $0.011236^{0.125} = 1.001398$ to $0.011236^{0.29} = 1.003248$. In short, owing to a bigger table, performance increases by a factor of 1.001398 to 1.003248 by raising p from 0.89 to 0.90.

[1]This assumption is rather optimistic, as table 3-1 costs are measured on CM5, while the real Phish system uses ethernet. As the Phish system will move to an ATM switch soon, however, the above estimation will become more practical. Above all, the same kind of analysis can be applied to new cost values if a more accurate optimal p is needed for such system.

[2]I also reanalyze by incorporating the copy cost into calculation and still reach the same optimal p. The copy cost barely effects the overall performance because only two of the processors are busy during the copying process. Moreover, when p is high enough, processor removals frequently leads to table contractions, thus reshuffling, instead of copying.

Therefore, by going from $p = 0.89$ to $p = 0.90$, the net performance gain is a factor of $\frac{1.001398}{1.000335} = 1.001063$ to $\frac{1.003248}{1.000335} = 1.002913$. Thus, we still gain by raising p from 0.89 to 0.90 because the net performance gain is greater than 1.

Repeating the same calculation for p from 0 to 1, I find that $p = 0.95$ to 0.96 is optimal for the Phish system pattern of usage. Table 4-2 shows the relevant part of the calculation. Full results are displayed in graph in figure 4-4.

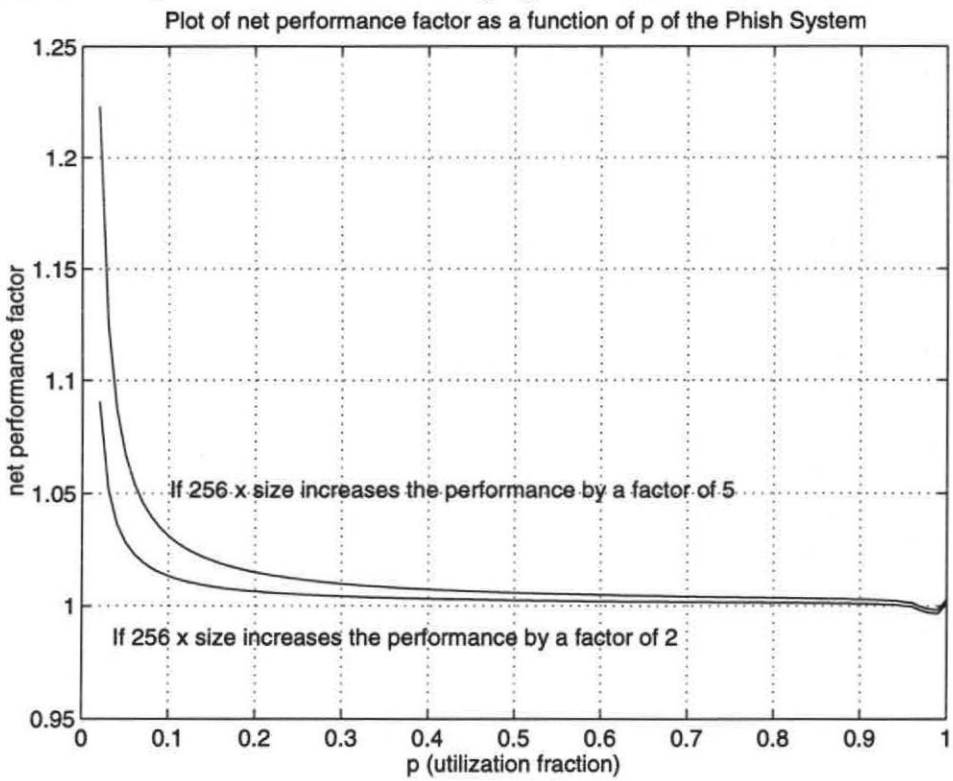
Table 4.2: The net performance difference moving one step from the preceding p to each p. The low and high gain factors are calculated if the performance increases by a factor of 2 and 5 successively when the table size grows 256 times. The number of reshuffles during 9.8 days of Phish is provided as a function of p.

p	the number of resuffles	the net performance difference using Hsu's low factor (2)	the net performance difference using Hsu's high factor (5)
0.86	934	1.001358	1.003295
0.87	1120	1.001226	1.003141
0.88	1252	1.001273	1.003166
0.89	1504	1.001115	1.002986
0.90	1787	1.001063	1.002913
0.91	2120	1.000988	1.002817
0.92	2504	1.000912	1.002721
0.93	3084	1.000665	1.002454
0.94	3745	1.000554	1.002324
0.95	4818	1.000050	1.001800
0.96	6169	0.999704	1.001436
0.97	8989	0.997940	0.999650
0.98	12741	0.996802	0.998493
0.99	16734	0.996480	0.998153
1.00	16734	1.001257	1.002921

4.4 Finding an Optimal p for the Phish System by Simulation

An optimal p can also be found experimentally by simulating a dynamic table behavior through a pattern of processor usage and accumulating work that would have been

Figure 4-4: Graph displays the net performance difference between successive p of the Phish system. Both low and high gain factors are provided.



accomplished. Using the performance model: $\Delta work = \Delta t \times size^{\alpha}$, where $\alpha = 0.125$, 0.29, and 1, figure 4-5 and figure 4-6 shows the work accomplished as a function of p for the random walk pattern and the Phish pattern of processor additions and removals. For the random walk pattern, an optimal p is 0.97, while for the Phish system, an optimal p ranges from 0.95 to 1, depending on α. The results agree with the analytical results found in the previous section.

Figure 4-5: Graph displays the work accomplished by the application program as a function of p, given a random walk pattern and the performance model of $\Delta work = \Delta t \times size^{\alpha}$, where α is 0.125, 0.29 and 1.

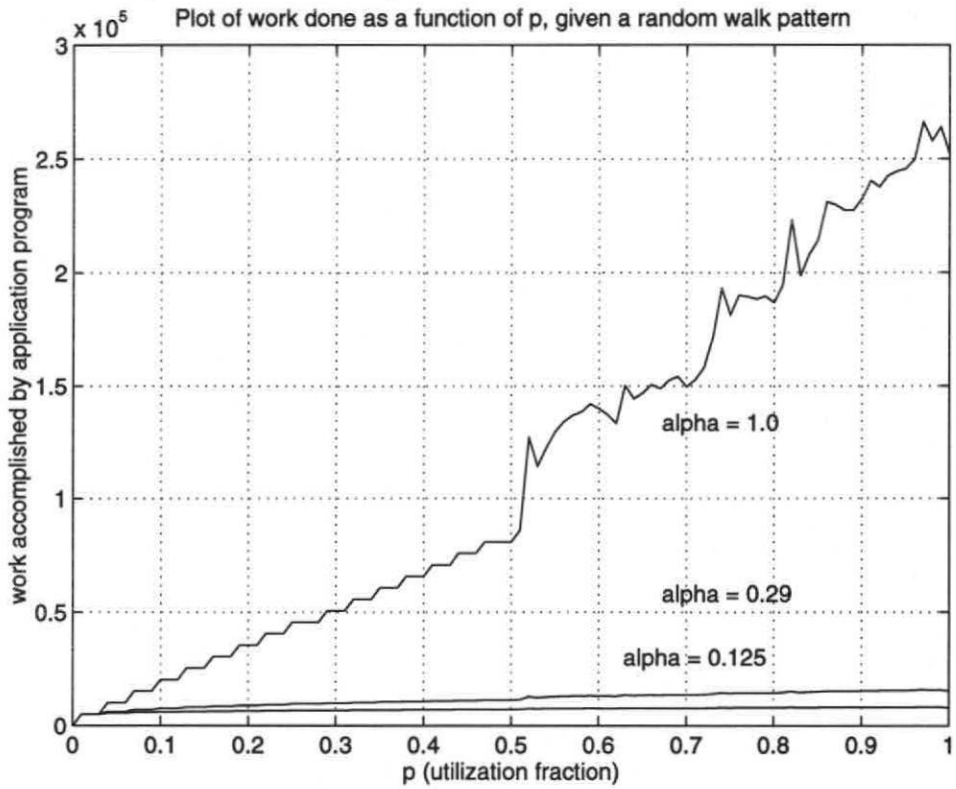
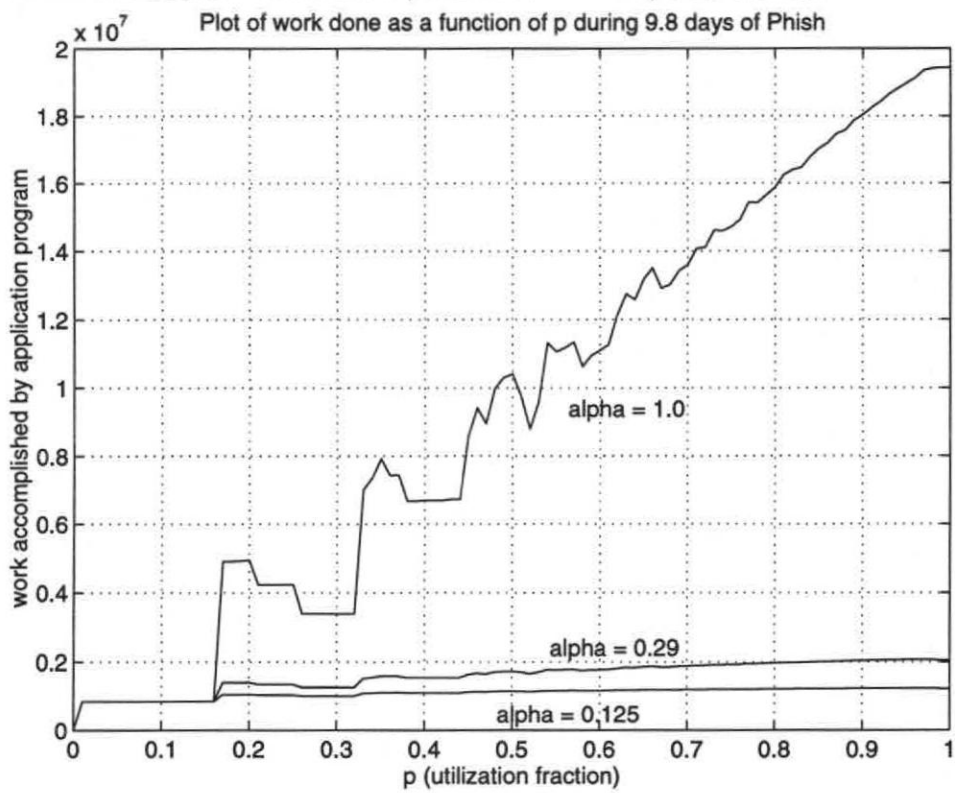


Figure 4-6: Graph displays the work accomplished by the application program as a function of p, given the Phish system pattern of processor usage and the performance model of $\Delta work = \Delta t \times size^{\alpha}$, where α is 0.125, 0.29 and 1.



Chapter 5

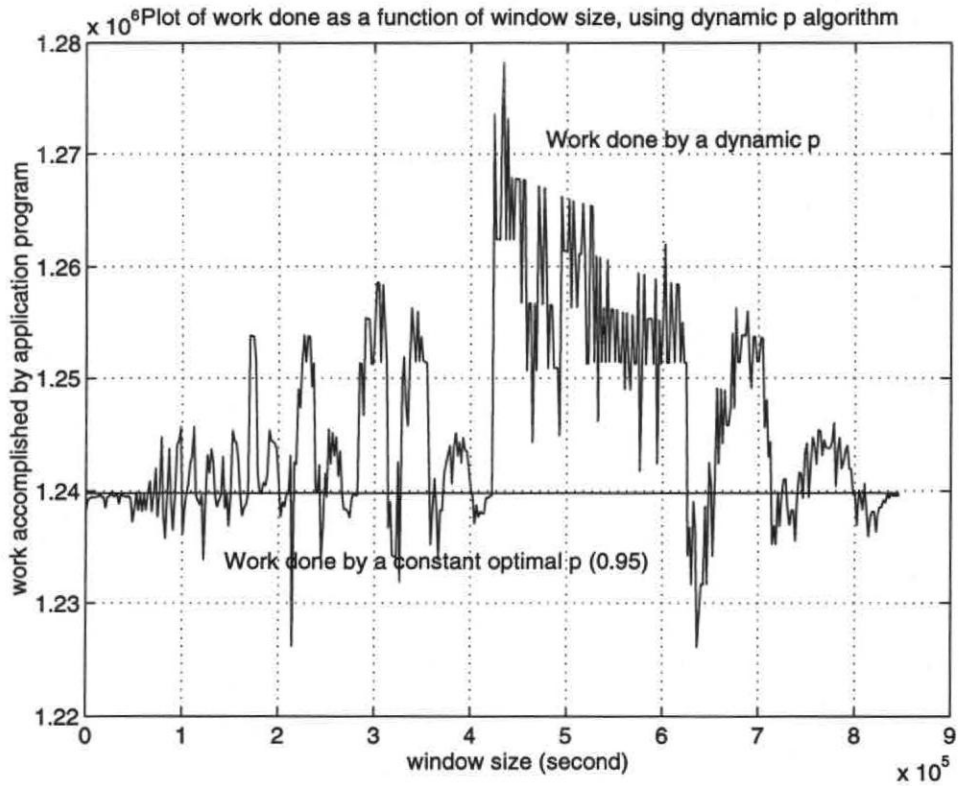
Adaptive Utilization Fraction

Previous chapters discuss the dynamic behavior of the table and how to derive p that renders the best performance for an application program. An optimal p can be calculated from the reshuffling cost, the value of the table size and the pattern of processor usage of a system. Without any of the information, we cannot derive an optimal p.

An alternative to achieve a maximum performance is to adjust p dynamically based on the performance history. The algorithm calculates the total work that would have been accomplished if p, p^2, and $\sqrt{p}$ were to be a utilization fraction during the past period, called *window*. It then uses the best result among the three to be a utilization fraction during the next window size. Since p is dynamically adjusted every window size, if the window size is sufficiently small, p will stay close to its optimal value at all time.

Figure 5-1 shows the result of the adaptive p algorithm on the Phish system. Regardless of the initial p values, the application performance under the adaptive p algorithm ranges from 99.6% to 103.1% when compared with the optimal result of a constant p. Notice that adaptive p algorithm yields the best performance when the window size is in the middle. When the window size is too small, the table becomes too sensitive to processor additions and removals and adjusts p too frequently. When the window size is too big, p approaches the optimal values slowly. Therefore, some performance is forfeited during the adjustment process.

Figure 5-1: Graph displays the work accomplished as a function of window size, using an adaptive p algorithm on the Phish system. The performance model is $\Delta work = \Delta t \times size^{0.125}$. An optimal work accomplished, using a constant p algorithm is provided for comparison. Note that most of the time, an adaptive p algorithm performs better than a constant p algorithm.



An adaptive p algorithm has several advantages over a constant p algorithm. First of all, an adaptive p algorithm requires no prior knowledge of processor usage pattern. Because performance is evaluated periodically, p automatically adapts to the current pattern of available memory. Secondly, the table can start with any initial p and still achieves the optimal results. A sufficient small window size ensures that p is adjusted frequently enough that it will approach the optimal value quickly. Thirdly, an adaptive p algorithm is likely to perform better than a constant p algorithm if the pattern of processor usage varies throughout time. For example, if the available memory stays constant for a period of time, an adaptive p algorithm will adjust p to 1 to utilize all the memory. Also, if the number of processors suddenly swings dramatically in a short time, an adaptive p algorithm will lower the p, thus, reduces the unnecessary reshuffling costs during that period. Because a constant p algorithm does not have this flexibility, it will function with the value of p that is best on average, but may not be the most obtainable.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Experimental results and data analysis have shown that a dynamic table on a parallel system is feasible for real applications. Even though there are several costs –namely a bookkeeping cost, a copy cost and a reshuffling cost – associated with maintaining the dynamicity of the table, such costs may be acceptable if they do not occur too often. The tradeoff lies in how much an application program gains from having a bigger table and how often the costs are incurred, given the pattern of the processor usage of the system.

Concretely, statistics have shown that if the Phish system communicates at the same speed as CM5, a chess program running on the Phish system should utilize up to 95% – 96% of the total processors to store the transposition table. Similar analysis can be applied to other systems to find an optimal utilization fraction of such systems, if their pattern of processor usage and the communication delay are known.

The project also suggests an adaptive p algorithm as an alternative to an optimal constant p analysis. Because the performance is evaluated periodically and p is adjusted accordingly, the overall performance of an application is likely to stay within an optimal range.

6.2 Future Work

Since the table costs are experimented on CM5, the gathered statistics are an optimistic view of costs. The Phish system, which currently uses an ethernet for communication, will require more time to reshuffle and copy entries from one processor to another. Therefore, should the chess program be ported onto the Phish system and uses a dynamic size transposition table, a precise measurement of the reshuffling and copy time on the Phish system will be necessary.

Moreover, my current implementation of the dynamic size table stores a mapping table on only one processor. Therefore, communication becomes critical, as all naming resolution between a virtual processor number and a real processor number has to be consulted with the central mapping table. Further research can explore the consequences and the feasibility of storing a mapping table on all or several processors. This approach allows processors to be added and removed, and entries to be inserted and looked up from any processors without communication delay. The reshuffle cost may also become lower, as the central mapping table no longer exists as a bottleneck. However, managing the consistency of the mapping tables may be more complicated and incur some other costs.

Lastly, as an adaptive p algorithm may potentially perform better than a constant p algorithm, further study can explore this algorithm more deeply. An implementation of the algorithm will reveal the issues, such as how to choose the window size, which processor usage pattern suits this algorithm, and the algorithm's complexity and feasibility on real applications.

Bibliography

[1] Robert D. Blumofe, Michael Halberr, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Phil Lisiecki, Keith H. Randall, Andy Shaw, and Yuli Zhou. *Cilk 1.1 Reference Manual.* Lab for Computer Science, MIT, 545 Technology Square, Cambridge, MA 02139, September 1994. Available via anonymous FTP from `theory.lcs.mit.edu` in `/pub/cilk/manual1.0.ps.Z`.

[2] Robert D. Blumofe and David S. Park. Scheduling large-scale parallel computations on networks of workstations. In *Proceedings of the Third International Symposium on High-Performance Distributed Computing (HPDC '94)*, San Francisco, California, August 1994. Available via anonymous FTP from `theory.lcs.mit.edu` in `/pub/rdb/hpdc94.ps.Z`.

[3] Feng hsiung Hsu. *Large Scale Parallelization of Alpha-Beta Search: An algorithmic and Architectural Study with Computer Chess.* Technical Report CMU-CS-90-108, Carnegie-Mellon University, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 15213, February 1990.

[4] Christopher F. Joerg and Bradley C. Kuszmaul. Massively parallel chess. In *Third DIMACS Parallel Implementation Challenge Workshop*, Rutgers University, October 1994. Available via anonymous FTP from `csg-ftp.lcs.mit.edu` in `pub/users/bradley/dimacs94.ps.Z`.

[5] Bradley C. Kuszmaul. *Synchronized MIMD Computing.* PhD dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering and

Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, June 1994.

[6] Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson. *Introduction to Algorithms.* The MIT Press and McGraw-Hill Book Company, The MIT Press Cambridge, Massachusetts London, England; McGraw-Hill Book Company New York St. Louis San Francisco Montreal Toronto, eighth printing edition, 1992.