# Efficient Parallel Formulations for Some Dynamic Programming Algorithms*

George Karypis and Vipin Kumar
karypis@cs.umn.edu and kumar@cs.umn.edu
Department of Computer Science
University of Minnesota
Minneapolis, MN 55414

TR 92-59, October 1992

**Abstract**

In this paper we are concerned with Dynamic Programming (DP) algorithms whose solution is given by a recurrence relation similar to that for the matrix parenthesization problem. Guibas, Kung and Thompson presented a systolic array algorithm for this problem that uses $O(n^2)$ processing cells and solves the problem in $O(n)$ time. In this paper, we present three different mappings of this systolic algorithm on a mesh connected parallel computer. The first two mappings use commonly known techniques for mapping systolic arrays to mesh computers. Both of them are able to obtain only a fraction of maximum possible performance. The primary reason for the poor performance of these formulations is that different nodes at different levels in the multistage graph in the DP formulation require different amounts of computation. Any adaptation has to take this into consideration and evenly distribute the work among the processors. Our third mapping balances the work load among processors and thus is capable of providing efficiency approximately equal to 1 (*i.e.* speedup approximately equal to the number of processors) for any number of processors and sufficiently large problem. We present a theoretical analysis of these mappings and experimentally evaluate them on a mesh embedded onto a 256 processor nCUBE/2*. It can be shown that our mapping can be used to efficiently map a wide class of two dimension systolic array algorithms onto mesh connected parallel computers.

*This work was supported by IST/SDIO through the Army Research Office grant #28408-MA-SDI and by the United States Army Research Office, Contract Number DAAL03-89-C-0038 at the University of Minnesota Army High Performance Computing Research Center.

*nCUBE/2 is a registered trademark of nCUBE Corporation.

# 1 Introduction

Dynamic programming (DP) is a widely used problem solving paradigm for optimization problems that is widely applied to a large number of areas including optimal control, industrial engineering, economics and artificial intelligence [3, 4, 14, 18]. Many practical problems involving a sequence of interrelated decisions can be efficiently solved by DP. The essence of many DP algorithms lies in computing solutions of the smallest subproblems and storing the results for usage in computing larger subproblems. Thus the solution to the original problem is constructed in a bottom-up fashion. DP problems can be represented graphically as a multistage graph where each node represents a subproblem. A node $u$ is connected to an other node $v$ with a directed arc if the subproblem $u$ is used in solving the subproblem $v$. A natural method of parallelizing various DP algorithms is to assign the task of of solving different subproblems to different processors.

A DP formulation is expressed as a recursive functional equation whose left-hand side is an expression involving the maximization (or minimization) of values of some cost functions. Li and Wah [11], have developed a classification of DP programming schemes according to the form of the functional equations and the nature of the recursion. A DP formulation is called *monadic* if its cost function involves only one recursive term, otherwise it is called *polyadic*. A DP formulation is *serial* if the subproblems can be grouped in levels, and the solution to any subproblem in a certain level can be found using subproblems that belong only in the immediately preceding levels, otherwise it is *non-serial*. As it was shown in [11, 5] *monadic-serial* DP problems can be solved by a series of matrix-vector multiplication which is easy to parallelize [5]. On the other hand there is no general parallel formulation for *polyadic-nonserial* DP problems.

In this paper we are concerned with the *polyadic-nonserial* DP algorithms whose solution is given by a recurrence relation similar to that for the matrix parenthesization problem [6]. Examples of these problems are: optimal triangularization of polygons, optimal binary search trees [6], and the CYK parser [1]. The serial complexity of these problems is $O(n^3)$. A number of parallel formulations have been proposed in [9] that use $O(n)$ processors on a hypercube and solves the problem in $O(n^2)$ time. A systolic array algorithm has been proposed in [7] that uses $O(n^2)$ processing cells and solves the problem in $O(n)$ time. Finally, there are some non-cost-optimal parallel formulations for PRAM machines that solve the problem in $O(\log^2 n)$ time using $\frac{n^6}{\log n}$ processors [15, 16].

The systolic algorithm for two dimension systolic arrays can be directly mapped onto a mesh connected parallel computer by assigning each cell to a different processor. This mapping leads to poor utilization, because in general purpose parallel computers, the communication cost for sending a unit message is much higher than unit computations. As shown in Ibara, Pong and Sohn [10], this problem can be corrected by assigning a block of of cells to each processor. Now the computation at each processor becomes proportional to the area of the block, and communication becomes proportional to the periphery. By choosing big enough block sizes, the ratio of communication to computation at each processor can be made arbitrarily small.

In this paper, we present three different formulations of the systolic algorithm [7] on a mesh connected parallel computer. The first formulation is a mapping of the systolic algorithm on a two dimension mesh computer along the lines proposed in [10]. This formulation results an upper bound on the efficiency equal to $\frac{1}{12}$ for sufficiently large number of processors. The second formulation is a slightly modified version of the first scheme but also has an upper bound of $\frac{1}{3}$ in efficiency. The primary reason for the poor performance of these formulations is that different nodes at different levels in the multistage graph require different amounts of computation. Any adaptation has to take this into consideration and evenly distribute the work among the processors. The third formulation uses a mapping that balances the work load among processors and thus is capable of providing efficiency approximately equal to 1 for any number of processors and sufficiently large problem. We present a theoretical analysis of these mappings and experimentally evaluate them on a mesh embedded onto a 256 processor nCUBE/2.

This paper is organized as follows: Section 2 and 3 present an overview of the dynamic programming algorithm and the available parallel formulations. Section 4 and 5 present and analyze our various mappings of the systolic algorithm onto a mesh parallel computer. Section 6 presents experimental results, and finally Section 7 provides some concluding remarks.

## 2 The Parenthesization Problem and the Dynamic Programming Algorithm

The parenthesization and other isomorphic problems, can be efficiently solved using a dynamic programming algorithm [6]. Consider the evaluation of the product of $n$ matrices, $A_1, A_2, \ldots A_n$, where each $A_i$ is a matrix with $r_{i-1}$ rows and $r_i$ columns. The order in which the matrices are multiplied together can have a significant effect on the total number of operations required to evaluate the product. Trying all possible orderings in which to evaluate the product of $n$ matrices so as to minimize the number of operations is an exponential process which is impractical when $n$ is moderately large. However, dynamic programming provides an $O(n^3)$ algorithm.

Let $c(i,j)$ be the cost of multiplying the matrices $A_i, A_{i+1}, \ldots A_j$. The dynamic programming paradigm constructs the solution to this problem based on the solution of its subproblems. This approach gives rise to the following recurrence relation for the parenthesization problem:

$$c(i,j) = \begin{cases} \min\limits_{i \leq k < j} \{c(i,k) + c(k+1,j) + r_{i-1} r_k r_j\} & 1 \leq i < j \leq n \\ 0 & j = i, 0 \leq i \leq n \end{cases} \tag{1}$$

Given equation (1) the problem reduces to finding the value for $c(1,n)$.

The solution to this recurrence relation, equation (1), is obtained by a bottom-up approach. An auxiliary table $C[n][n]$ is used for storing the values of $c(i,j)$ and an other one $S[n][n]$ for storing the optimal indices for $k$. The algorithm fills in the tables $C$ and $S$ in a manner that corresponds to solving the parenthesization problem on matrix chains of increasing length. We can graphically visualize this if we think of filling in the tables in a diagonal order (see Figure 1). This concept of diagonal oriented computations will be extensively used in the rest of this paper. For a more detailed description refer to [6].

Computing entry $c(i,j)$ requires computing the cost of $(j-i)$ possible parenthesizations and taking their minimum. This can be done in $(j-i)$ operations. Hence, the exact sequential complexity of the algorithm is: $\sum_{i=1}^{n-1}(n-i)i = \frac{n^3}{6} - \frac{n}{6}$. For the rest of this paper, in order to simplify the computations, we will assume that elements of the $k^{th}$ diagonal require $k$ computations, where $1 \leq k \leq n$. For this case, the exact sequential complexity of the algorithm is: $\sum_{i=1}^{n}(n-i+1)i = \frac{n^3}{6} + \frac{n^2}{2} + \frac{n}{3}$, which is approximately $\frac{n^3}{6}$ for large enough $n$.

## 3 Parallel Formulations of the Dynamic Programming Algorithm

The dynamic programming algorithm for the parenthesization problem can be easily parallelized using a linear array of $p$ processors where $1 \leq p \leq n$. This linear array formulation will compute successive diagonals of matrix $C$ at successive steps. If there are $I$ nodes in a diagonal, we assign $I/p$ nodes to each of the $p$ processors. Each processor computes the cost of the entries $c(i,j)$ assigned to it. This is followed by an all-to-all broadcast [5] during which solution costs of the subproblems at that diagonal are made known to all the processors. Since each processor has complete information about subproblem costs at preceding diagonals, no communication is needed other than the all-to-all broadcast. The cost of performing the all-to-all broadcast of $O(\frac{n}{p})$ information among $p$ processors is $O(n)$ hence, The runtime of this formulation is $O(\frac{n^3}{p}) + O(n^2)$, where $O(\frac{n^3}{p})$ is the time spent in computation, and $O(n^2)$ communication time. If $n$ is sufficiently larger than $p$, then the communication time can be made to be an arbitrarily small fraction of the computation time, and linear speedups can be obtained. An alternative mapping was proposed by Ibara , Pong and Sohn in the context of the CYK parser [9]. Their formulation uses $p = O(n)$ processors, connected in a hypercube topology, and solves the problem in $O(\frac{n^3}{p})$ time, which is cost optimal. The formulation of Ibara *et.al.* has properties similar to the formulation for linear array mentioned above. Both formulations are efficient only if $p$ is sufficiently smaller than $n$.

A faster formulation can be achieved using $\frac{n(n+1)}{2}$ processors on a PRAM machine. In this mapping each processor computes an entry $c(i,j)$ of the matrix $C$. From equation 1, it can be shown that having finished diagonal $t$, we can perform some computations on the subsequent $t{+}1$ diagonals. Thus, the work in diagonal $n$ can start when diagonal $\frac{n}{2}$ has been computed. Furthermore, we know that entries in diagonal $n$ require $n$ computations; hence, the runtime of this formulation is given by the recurrence relation: $T(n) = T(\frac{n}{2}) + n$, whose solution for sufficiently large $n$ is: $T(n) = 2n$. The exact processor–time product of the PRAM formulation is $\frac{n(n+1)}{2} \times 2n \approx n^3$; hence, even though the PRAM algorithm is significantly faster, it does 6 times more work than the sequential algorithm therefore its efficiency is only 0.167.

Guibas, Kung and Thomson [7] have developed a systolic algorithm for the parenthesization problem. Their algorithm uses $\frac{n(n+1)}{2}$ processing elements (cells) connected as a two dimension systolic array (TSA) as shown in Figure 1, and solves the problem in in essentially the same time as the PRAM algorithm outlined above. For the rest of this paper we will refer to this algorithm as GKT. A brief description of the algorithm follows. For a more detailed description the reader should refer to [7].

The inputs $c(i,i)$ are applied in parallel to the cells with coordinates $(i,i)$ and each cell $(i,j)$ computes $c(i,j)$. If a cell is computing an element of diagonal $t$, then its result is ready at time $2t$. At that moment the cell starts transmitting its result upwards and to the right. The result travels along both directions by moving by one cell per time unit for $t$ additional units. From that moment until eternity the result moves a cell every two time units. During a time unit a cell $(i,j)$ will receive results for previous subproblems. If the new results improve the cost, they replace the currently held values. At each time unit a cell receives at most two sets of results from smaller subproblems, hence it has to perform at most two sets of computations.

The purpose of this paper is to investigate a number of possible mappings of the GKT algorithm onto a mesh connected parallel computer [2] with $p$ processors. A mesh connected parallel computer has a structure similar to that of the TSA hence, the mapping of the TSA algorithm onto a mesh can be done in a natural way. Furthermore, we will assume that the mesh connected parallel computer has wrap-around communication links. This is done merely to simplify the presentation and is not required by any of our proposed mappings.

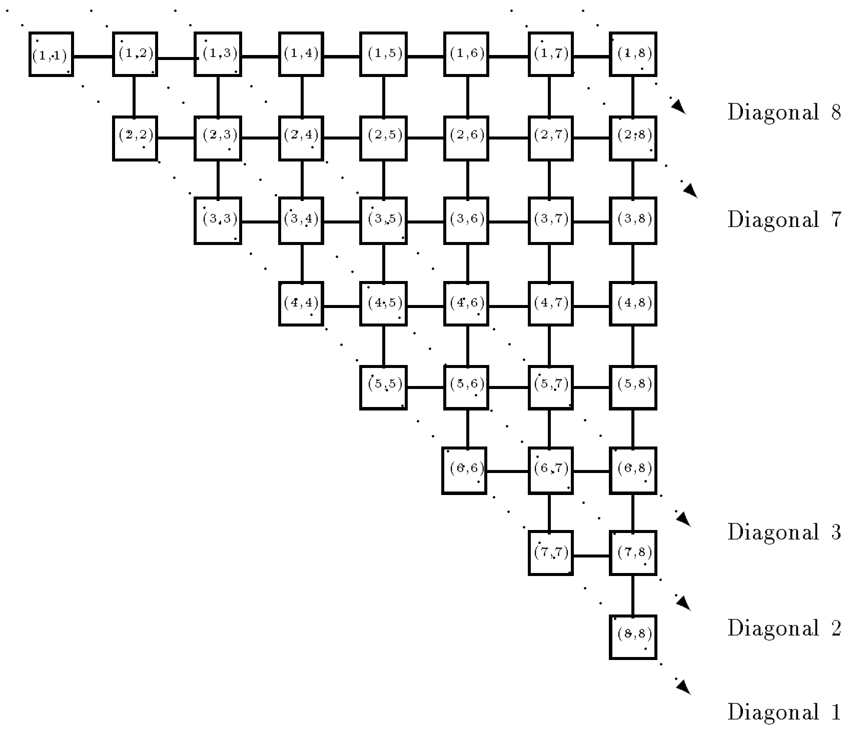


Figure 1: GKT algorithm for a TSA

# 4 Mapping the Systolic Algorithm onto a Mesh Parallel Computer

In the GKT algorithm as described in Section 2, results are communicated at two different speeds (either once every time unit or twice every time unit). This guarantees that results arrive at a cell when this cell is ready to use them. This is important for systolic algorithms, as a systolic array is supposed to have only a small amount of memory at each cell. General purpose processors have substantial amounts of memory which can be used to store results arriving at earlier times. In our mappings, messages are transmitted with no delays. When messages are received at a processor they are stored in local memory until they are used.

Despite the above simplification, mapping the GKT algorithm onto a mesh connected parallel computer poses a number of problems. Direct implementation of the systolic algorithm (*i.e.* use of $\frac{n(n+1)}{2}$ processors) will lead to an inefficient algorithm and underutilization of the parallel computer. This is because in general purpose parallel computers, the cost of sending an element to another processor is much higher than the cost of performing the computations associated with that element. For example in nCUBE/2, the cost of sending one element is $180\mu s$ while the cost of performing a computation is $4\mu s$ for the parenthesization problem.

A solution to this problem is to map more than one TSA cell onto a single mesh processor. The computations associated with each processor, in a time step, is usually proportional to the number of cells assigned to it, while the communication is proportional to the number of cells it has at the boundary. For example if we map $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ cells on each processor then the computations is proportional to $\frac{n^2}{p}$ while the communication is proportional to $\frac{n}{\sqrt{p}}$. By varying the number of processors, we can adjust the cost of communication to computation and hence obtain an efficient parallel formulation.

Furthermore, an efficient mapping has to keep as many processors doing useful work as possible. Due to the nature of the GKT algorithm, the computations will move in a wavefront form within the TSA. At any given time in the execution of the algorithm, just a band of diagonal cells will be performing computations, while the remaining cells will either have finished their share of work or will be waiting to receive results for subproblems currently being computed. This computational pattern will lead to cells sitting idle at various points of the execution of the algorithm, and depending on the mapping might lead to processors sitting idle as well.

Finally, different cells in the GKT algorithm will perform different amounts of computation. Each TSA cell will compute an entry $c(i,j)$. The amount of computation required is proportional to the diagonal that $c(i,j)$ belongs to. The higher the diagonal (*i.e.* greater the value of $(j-i)$), the higher the amount of required computation. Hence, even though we might be able to map the same number of cells onto each processor, the computations required may vary significantly.

Given these criteria for efficient mappings of the GKT algorithm onto a mesh connected parallel computer, we present and analyze three different mappings that address these issues to different degrees. We assume that the mesh has $\sqrt{p} \times \sqrt{p}$ processors, and $n$ is a multiple of $\sqrt{p}$. These mappings are described in the following sections.

## 4.1 Checkerboarding Mapping (CM)

An intuitive and straight forward mapping is to group blocks of $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ cells together and map them onto the same processor. In particular, let $P_{i,j}$ be the processor of the $i^{th}$ row and $j^{th}$ column of the mesh, where $1 \leq i \leq j \leq \sqrt{p}$. Each $P_{i,j}$ processor will compute the entries $c(y,x)$ such that: $(i-1)\frac{n}{\sqrt{p}} + 1 \leq x \leq i\frac{n}{\sqrt{p}}$ and $(j-1)\frac{n}{\sqrt{p}} + 1 \leq y \leq j\frac{n}{\sqrt{p}}$. Because of the triangular shape of the TSA, processors $P_{i,j}$ for $i > j$ will not be assigned any TSA cells while, the processors $P_{i,i}$ will be assigned only $\frac{n}{2\sqrt{p}}(\frac{n}{\sqrt{p}} + 1)$ cells. Figure 2 illustrates this mapping. This mapping scheme is often called checkerboarding and it has been used in a number of applications [8, 5, 10]. Each processor has to store at most $\frac{n}{\sqrt{p}}$ rows and $\frac{n}{\sqrt{p}}$ columns; hence, the memory requirements at each processor is $O(\frac{n^2}{\sqrt{p}})$.

The only TSA cells that need to communicate with the surrounding processors are those along the periphery of the block where for every diagonal received or computed each processor can perform computations on a number of diagonals residing on it. Hence, the computation performed is $O(\frac{n^2}{p})$ while the communication is $O(\frac{n}{\sqrt{p}})$.

However, checkerboarding mapping has a number of limitations. It maps cells only to $\frac{\sqrt{p}(\sqrt{p}+1)}{2}$ processors, and thus the remaining $\frac{\sqrt{p}(\sqrt{p}-1)}{2}$ processors are idle all the time. Also due to the nature of the algorithm, the band of diagonal entries being computed will reside on a small number of adjacent diagonals of mesh processors. For example, after computing diagonal $t$ we can perform computations on the following $\min(t+1, n-t)$ diagonals. Hence, during that time processors on these diagonals processors will be performing computations while the remaining processors will either have finished their work or will be waiting to receive diagonals that are currently being computed. Finally, because computations associated with a cell increases as the number of the diagonal containing this cell increases, different processors will have different work loads even though they have the same number of TSA cells.

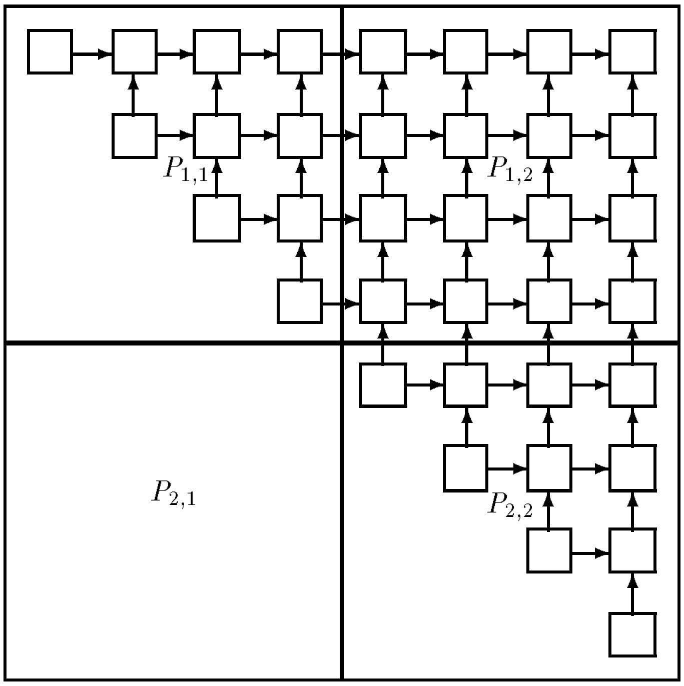


Figure 2: Checkerboarding Mapping

### 4.2 Modified Checkerboarding Mapping (MCM)

One of the limitations of the checkerboarding mapping is that it maps no work to about half of the processors. The modified checkerboarding solves this problem and at the same time it preserves the communication properties of CM.

The modified checkerboarding mapping is achieved as follows: First we partition the TSA into 3 blocks $L_1, L_2$ and $L_3$ as illustrated in Figure 3. Block $L_3$ is partitioned in a checkerboarding fashion into blocks each having $\frac{n}{2\sqrt{p}} \times \frac{n}{2\sqrt{p}}$ cells, and is mapped onto the processor mesh. Blocks $L_1$ and $L_2$ are rotated $180^o$ about their common boundary with block $L_3$ and then are mapped onto the processor mesh in a checkerboarding fashion. An alternative way of visualizing this mapping is to think that first the TSA is being folded along the common boundaries $L_1 - L_3$ and $L_2 - L_3$ and then the $\frac{n}{2} \times \frac{n}{2}$ square obtained, is mapped onto the $\sqrt{p} \times \sqrt{p}$ processor mesh in a checkerboarding fashion. Similarly to checkerboarding, the memory requirements by each processor is $O(\frac{n^2}{\sqrt{p}})$.

This mapping guarantees that all the processors will have the same number of cells $\frac{n^2}{2p}$, with the exception of the processors $P_{i,\sqrt{p}+1-i}$ for $1 \leq i \leq \sqrt{p}$ that have $\frac{n}{2\sqrt{p}}$ additional cells. Also, it can be easily seen that the work mapped to processors on the same row is roughly the same (with the exception of the diagonal processors). The same statement though, doesn't hold for the processors along the same column. The processors at the first row do more work than those at the second, and so forth. For example, each cell of a processor at the first row belongs to a diagonal that is by $\frac{n}{\sqrt{p}}$ higher than the corresponding cell of the processor at the second row. Thus, the processors at the first row has to perform $\frac{n^3}{p\sqrt{p}}$ more computations. Similarly, the processors at the second row has to perform $\frac{n^3}{p\sqrt{p}}$ more computations than the processors at the third row, and so forth. Hence, even

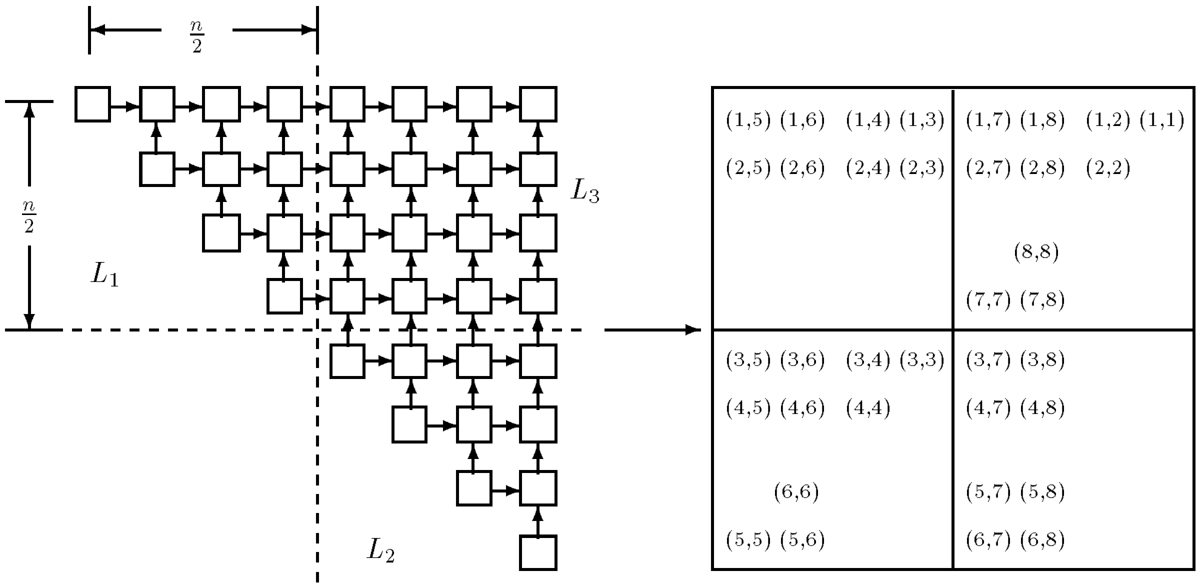


Figure 3: Modified Checkerboarding Mapping

though modified checkerboarding utilizes $p$ processors, it does not eliminate work load imbalances.

### 4.3 Shuffling Mapping

The two mappings proposed so far didn't fully address the various issues involved in efficiently mapping the TSA algorithm onto a mesh connected parallel computer. Even though communication locality was a property of both the checkerboarding and the modified checkerboarding mappings, work load was unevenly distributed among the processors. Here we present a different mapping that preserves the communication characteristics of checkerboarding and at the same time evenly distributes the work among the processors.

This new mapping maps successive rows and columns of cells onto successive rows and columns of the mesh respectively. In particular, the $c(i,j)$ cell of the TSA is mapped onto the $(((i-1) \bmod \sqrt{P})+1, ((j-1) \bmod \sqrt{P})+1)$ processor of the mesh. The above definition requires a wrap around mesh but there is an alternative way of mapping the TSA that eliminates this requirement. We can think of the TSA as being partitioned into columns each containing $\sqrt{p}$ consecutive cells. Then these 'fat' columns are being folded along their common boundaries and a column containing $\sqrt{p}$ cells of depth $\frac{n}{\sqrt{p}}$ is obtained. This column is again being partitioned into rows each having $\sqrt{p}$ cells and these rows are being folded along their common boundaries. The resulting $\sqrt{p} \times \sqrt{p}$ block is then mapped onto the processor mesh. This mapping maps either $\frac{n}{\sqrt{p}}(\frac{n}{\sqrt{p}}+1)$ or $\frac{n}{\sqrt{p}}(\frac{n}{\sqrt{p}}-1)$ TSA cells onto a mesh processor. Even though adjacent rows and columns of the TSA are being mapped onto adjacent rows and columns of the processor mesh, the amount of communication performed is similar to the checkerboarding scheme. This is because when cell $c(i,j)$ sends its results to cell $c(i,j+1)$ then the result is also received by cells $c(i,j+1+k\sqrt{p})$ for $k = 1,2,3,\ldots$. This mapping is illustrated in Figure 4. For the rest of this paper this mapping will be referred to as shuffling. A variation of this mapping was used in the context of shortest path on sparse graphs in [17]. The memory requirements by each processor is $O(\frac{n^2}{\sqrt{p}})$.

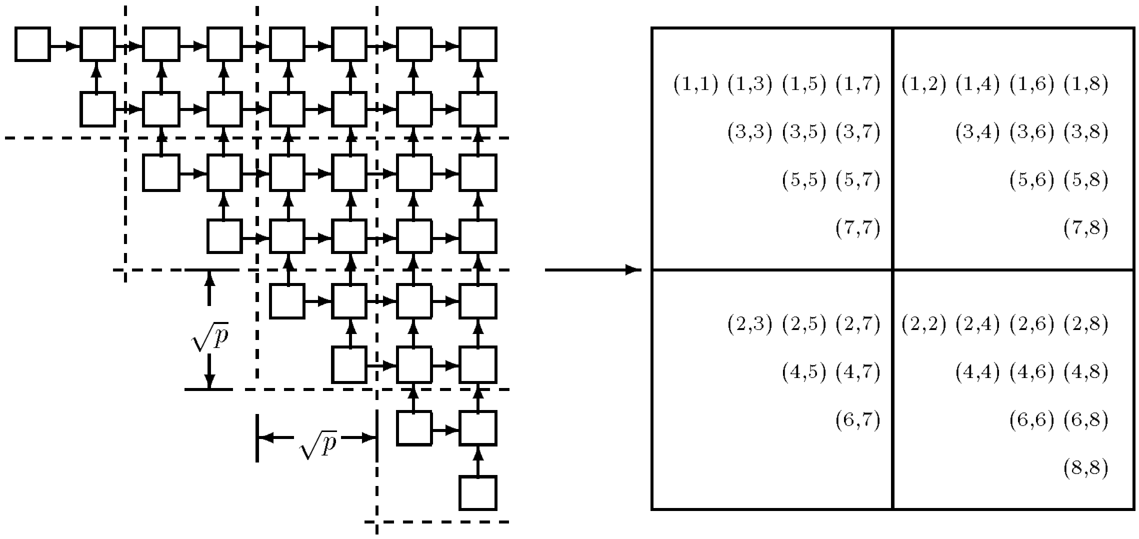


Figure 4: Shuffling mapping of a $8 \times 8$ TSA onto a $2 \times 2$ mesh.

Note that both in CM and MCM mappings, each processor is assigned portions of consecutive diagonals where in SM each processors is assigned portions of diagonals that are $\sqrt{p}$ apart. As we know, the amount of work required to compute a diagonal increases as the diagonal increases; thus, in the CM and MCM mappings the processors having higher diagonals will do more work than those having lower ones. On the other hand, in SM, each processor is assigned an equal number of low and high diagonals, thus the work allocated to each processor doesn't vary significantly. Furthermore, because consecutive TSA rows and columns reside on consecutive mesh rows and columns, the processors will start working at an earlier time compared with either checkerboarding or modified checkerboarding mappings.

# 5 Analysis of the Various Mappings

## 5.1 Definitions and Assumptions

This section contains various definitions used in the presentation and the analysis of the various mappings.

- Problem size $W$: the amount of computation performed by the serial algorithm. $W \approx \frac{n^3}{6}$.
- Input size $n$: the number of matrices that the optimal parenthesization needs to be found.
- Number of processors $p$: the number of identical processors in the mesh available to solve the problem.
- Unit computation time $t_c$: the time taken for one unit of work. A unit of work is defined as the time to calculate the cost of combining two subproblems and comparing the results against the current minimum.

- Unit message transfer time $t_w$: the time required to send one piece of data between to adjacent processors. A piece of data is defined to be the solution $c(i, j)$ of a subproblem. If $m$ pieces of data is sent then the transfer time is $mt_w$.

- Message startup time $t_s$: the time required to setup a message transfer operation. The startup time is independent of the size of the transmitted message.

- Communication time $T_{comm}^{P_{i,j}}$: is the time spent by processor $P_{i,j}$ in performing communication and idling. This is the time spent in sending and waiting for messages, and for possible idling at the beginning of local computations.

- Computation time $T_{calc}^{P_{i,j}}$: is the time spent by processor $P_{i,j}$ in performing computations.

- Parallel run time $T_{par}$: the execution time on a $P$ processor mesh. Note that $T_{par} = T_{calc}^{P_{i,j}} + T_{comm}^{P_{i,j}}$ for any processor $P_{i,j}$.

- Speedup $S$: the ratio $\frac{W}{T_{par}}$.

- Efficiency $E$: is the speedup divided by the number of processors. $E$ denotes the effective utilization of computing resources.

We assume that every time a message is send, the sending processor pays only the cost of setting up the various parameters involved (*i.e.* message startup time $t_s$) and it doesn't pay the cost of the message transition time. Also, the receiving processor can proceed working while the message is being transmitted. These assumptions hold for most current generation parallel computers such as nCUBE/2.

### 5.2 Scalability Analysis Using the Isoefficiency Function

If a parallel algorithm is used to solve a problem instance of a fixed size, then the efficiency decreases as the number of processors $P$ increases. The reason is that the total overhead increases with $P$. For many parallel algorithms, for a fixed $P$, if the problem size $W$ is increased, then the efficiency becomes higher because the total overhead grows slower than $W$. For these parallel algorithms, the efficiency can be maintained at a desired level with increasing number of processors, provided the problem size is also increased. We call such algorithms **scalable** parallel algorithms.

For a given parallel algorithm, for different parallel architectures, the problem size may have to increase as a different function of $P$ in order to maintain a fixed efficiency. The rate that $W$ has to increase as a function $P$ to keep the efficiency fixed is essentially what determines the degree of scalability of the algorithm architecture combination. If $W$ has to increase as an exponential function of $P$, then the algorithm-architecture combination is poorly scalable. The reason for this is that in this case it would be difficult to obtain good speedup on the architecture for a large number of processors, unless the problem size being solved is enormously large. On the other hand if $W$ needs to grow linearly as a function of $P$ then the algorithm-architecture combination is highly scalable and can easily deliver linearly increasing speedup with increasing number of processors for reasonable increments of problem sizes. If $W$ needs to grow as $f_E(P)$ to maintain an efficiency $E$, then $f_E(P)$ is defined to be the **isoefficiency function** for efficiency $E$ and the plot of $f_E(P)$ with respect to $P$ is defined to be the **isoefficiency curve** for efficiency $E$.

A lower bound on any isoefficiency function is that asymptotically, it should be at least linear. This follows from the fact that all problems have a sequential (*i.e.* non decomposable) component.

Hence any algorithm which shows a linear isoefficiency on some architecture is optimally scalable on that architecture. For the parenthesization problem, the lower bound on the isoefficiency is $O(p^{1.5})$. This is because the TSA algorithm requires $O(n^2)$ processors ($i.e$ $n^2 \sim p \Rightarrow n \sim \sqrt{p} \Rightarrow W \sim p^{1.5}$). For a more rigorous discussion on the isoefficiency metric and scalability analysis, the reader is referred to [13, 12].

### 5.3 Analysis of Checkerboarding Mapping

In analyzing the performance of the checkerboarding mapping we will concentrate on the amount of computation $T_{calc}^{P_{1,\sqrt{p}}}$, performed by processor $P_{1,\sqrt{p}}$ because this processor performs more computation than any other processor. Clearly, $T_{calc}^{P_{1,\sqrt{p}}}$ is a lower bound of the runtime of the parallel formulation.

Processor $P_{1,\sqrt{p}}$ will compute elements from the last $(2\frac{n}{\sqrt{p}} - 1)$ diagonals. Note that each cell along the $(\sqrt{p}-1)\frac{n}{\sqrt{p}} + 1$ diagonal requires $(\sqrt{p}-1)\frac{n}{\sqrt{p}} + 1$ computations where cells below this diagonal require less work, and vice-versa. It is easily seen that the sum of the work of any two cells that are symmetrically located across this diagonal is equal to twice the work of a cell on diagonal $(\sqrt{p}-1)\frac{n}{\sqrt{p}} + 1$. Because there are $\frac{n^2}{p}$ cells in processor $P_{1,\sqrt{p}}$, the total amount of computation performed by this processor is:

$$T_{calc}^{P_{1,\sqrt{p}}} = \frac{n^2}{p}((\sqrt{p}-1)\frac{n}{\sqrt{p}} + 1)t_c = \frac{n^2}{p}(n - \frac{n}{\sqrt{p}} + 1)t_c = \frac{n^3}{p}t_c - \frac{n^3}{p\sqrt{p}}t_c + \frac{n^2}{p}t_c \tag{2}$$

From this equation we can see that the computations performed by processor $P_{1,\sqrt{p}}$ is $\frac{n^3}{p}t_c$ (ignoring lower order terms[1]). The sum of the work done by all the processors is at least $n^3 t_c$, which is six times more than the computations performed by the serial algorithm. Due to this excess computations the efficiency of the parallel formulation is bounded by:

$$E < \frac{\frac{n^3}{6}t_c}{n^3 t_c} = \frac{1}{6} = 0.167 \tag{3}$$

In practice, the parallel runtime of the algorithm will be much higher than $T_{calc}^{P_{1,\sqrt{p}}}$ due to a) communication costs and b) idling during the beginning of the algorithm. It is shown in Appendix B that assuming there are no communication costs ($i.e.$ considering only the overheads due to idling), the parallel runtime of the checkerboarding mapping is:

$$\left(\frac{n^3}{p}\left((\sum_{i=0}^{(\log\sqrt{p})-1}\frac{1}{2^i}) - \frac{\log\sqrt{p}}{\sqrt{p}}\right) - \frac{n^3}{3p\sqrt{p}}((\log\sqrt{p}) - 1)\right)t_c$$

Hence, for sufficiently large $p$, this expression reduces to $\frac{2n^3}{p}t_c$, and thus the upper bound in efficiency is only $\frac{1}{12}$. We have computed the upper bound in the efficiency for different number of processors using this equation, and the results are shown in Table 1.

### 5.4 Analysis of Modified Checkerboarding Mapping

Similarly to the CM case, we will concentrate our analysis on the amount of computation $T_{calc}^{P_{1,\sqrt{p}}}$, performed by processor $P_{1,\sqrt{p}}$, as this will be a lower bound of the runtime of this parallel formulation.

[1]Because $O(n^2)$ processors are being used, $\frac{n^3}{p\sqrt{p}} = O(1)$ and $\frac{n^2}{p} = O(1)$ where $\frac{n^3}{p} = O(n)$

| $p$ | 4 | 16 | 64 | 256 | 1024 | 4096 |
|---|---|---|---|---|---|---|
| $E$ | 0.33 | 0.17 | 0.13 | 0.10 | 0.09 | 0.083 |

Table 1: Efficiency upper bounds for the checkerboarding mapping.

For the modified checkerboarding mapping processor $P_{1,\sqrt{p}}$ is assigned work from all three blocks $L_1$, $L_2$ and $L_3$. Due to symmetry the work from block $L_1$ and $L_2$ is the same and is: $\sum_{i=1}^{\frac{n}{2\sqrt{p}}} i(\frac{n}{2\sqrt{p}} + 1 - i)t_c$. The work from block $L_3$ from a similar analysis to that for the checkerboarding mapping is: $\frac{n^2}{4p}(n - \frac{n}{2\sqrt{p}} + 1)t_c$. Hence, the work assigned to processor $P_{1,\sqrt{p}}$ is:

$$\begin{aligned} T_{calc}^{P_{1,\sqrt{p}}} &= \left( 2\sum_{i=1}^{\frac{n}{2\sqrt{p}}} i(\frac{n}{2\sqrt{p}} + 1 - i)t_c \right) + \frac{n^2}{4p}(n - \frac{n}{2\sqrt{p}} + 1)t_c \\ &= \frac{n^3}{4p}t_c + \frac{n^3}{16p\sqrt{p}}t_c + \frac{5n^2}{12p}t_c + \frac{n}{6\sqrt{p}}t_c + \frac{2}{3}t_c \end{aligned} \tag{4}$$

From this equation we can see that the computations performed by processor $P_{1,\sqrt{p}}$ is: $\frac{n^3}{4p}t_c$ (ignoring lower order terms); hence, the modified checkerboarding mapping performs 1.5 times more computations than the optimal formulation. Due to this excess computations the efficiency of this parallel formulation is bounded by:

$$E = \frac{\frac{n^3}{6}}{\frac{n^3}{4}} = \frac{2}{3} = 0.667 \tag{5}$$

Similarly to the checkerboarding mapping, equation (4) is just a lower bound and the actual runtime will be higher due to communication costs and idling. It is shown in Appendix C that assuming there are no communication costs (*i.e.* considering only the overheads due to idling), the parallel runtime of the modified checkerboarding mapping is:

$$\left( \frac{n^3}{4p} \left( (\sum_{i=0}^{\log\sqrt{p}} \frac{1}{2^i}) - \frac{\log 2\sqrt{p}}{2\sqrt{p}} \right) - \frac{n^3}{24p\sqrt{p}}(\log\sqrt{p}) \right) t_c$$

Hence, for sufficiently large $p$, this expression reduces to $\frac{n^3}{2p}t_c$, and thus the upper bound in efficiency is only $\frac{1}{3}$. We have computed the upper bound on the efficiency for different number of processors using this equation, and the results are shown in Table 2. Even though, the modified checkerboarding mapping is an improvement over the checkerboarding mapping, it is unable to provide efficiency close 1 even for arbitrary large $n$. As mentioned in Section 4.2, this is due to work imbalances among successive rows of mesh processors.

### 5.5 Analysis of Shuffling Mapping

As we did in the analysis of the previous two mappings in analyzing the performance of shuffling we will again concentrate on processor $P_{1,\sqrt{p}}$. Processor $P_{1,\sqrt{p}}$ will calculate cells that belong in the

| $p$ | 4 | 16 | 64 | 256 | 1024 | 4096 |
|---|---|---|---|---|---|---|
| $E$ | 0.72 | 0.53 | 0.43 | 0.39 | 0.36 | 0.33 |

Table 2: Efficiency upper bounds for the modified checkerboarding mapping.

diagonals $\sqrt{p}, 2\sqrt{p}, \ldots, n$. The computation performed by this processor $T_{calc}^{P_{1,\sqrt{p}}}$ is:

$$\begin{aligned} T_{calc}^{P_{1,\sqrt{p}}} &= t_c \sum_{i=1}^{\frac{n}{\sqrt{p}}} (\frac{n}{\sqrt{p}} + 1 - i) i \sqrt{p} \\ &= \frac{n^3}{6p} t_c + \frac{n^2}{2\sqrt{p}} t_c + \frac{n}{3} t_c \end{aligned} \tag{6}$$

It can be similarly shown, that the computations performed by any processor $P_{i,j}$ is: $T_{calc}^{P_{i,j}} = T_{calc}^{P_{1,\sqrt{p}}} - (i - 1 + \sqrt{p} - j)\frac{n^2}{p}$. Thus, in the worst case the computations performed by any other processor will be smaller only by $O(\frac{n^2}{\sqrt{p}})$. This imbalance becomes insignificant compared with the work done by each processor as $n$ increases. Nevertheless, in order to prove that shuffling yields a parallel formulation capable of achieving efficiency of 1, we must compute the amount of time spent by processor $P_{1,\sqrt{p}}$ in performing communications.

Computing $T_{comm}^{P_{1,\sqrt{p}}}$ is more complicated than computing $T_{calc}^{P_{1,\sqrt{p}}}$ because the algorithm overlaps computations with communication. From Appendix A, we have that:

$$T_{comm}^{P_{1,\sqrt{p}}} < (\frac{n^2}{\sqrt{p}} + n) t_w + (\frac{n^2}{\sqrt{p}} + n) t_c + 2 n t_s \tag{7}$$

From equation (6) and equation (7) we have that an upper bound on the the parallel runtime of the algorithm is:

$$\begin{aligned} T_{par} &< T_{comm}^{P_{1,\sqrt{p}}} + T_{calc}^{P_{1,\sqrt{p}}} \\ &< (\frac{n^3}{6p} + \frac{3n^2}{2\sqrt{p}} + \frac{4n}{3}) t_c + (\frac{n^2}{\sqrt{p}} + n) t_w + 2 n t_s \end{aligned} \tag{8}$$

The efficiency of the algorithm is:

$$\begin{aligned} E &= \frac{\frac{n^3}{6} t_c}{(\frac{n^3}{6p} + \frac{3n^2}{2\sqrt{p}} + \frac{4n}{3}) p t_c + (\frac{n^2}{\sqrt{p}} + n) t p_w + 2 n p t_s} \\ &= \frac{1}{1 + (\frac{9\sqrt{p}}{n} + \frac{8p}{n^2}) + (\frac{6\sqrt{p}}{n} + \frac{6p}{n^2})\frac{t_w}{t_c} + \frac{12p}{n^2}\frac{t_s}{t_c}} \end{aligned} \tag{9}$$

From equation (9) we can see that as $n$ increases, the efficiency of the algorithm increases approaching 1; hence, shuffling yields a cost optimal parallel formulation. Also, from this equation it can be shown that the isoefficiency function is: $O(p^{1.5})$. This isoefficiency function is the same for both the overheads due to extra work and also the overheads due to communication. Hence, shuffling mapping can effectively utilize $O(n^2)$ processors.

# 6 Experimental Results

We implemented all three mappings of the GKT algorithm presented in Section 4 on an nCUBE/2 parallel computer. nCUBE/2 is a hypercube connected parallel computer and a well known mapping [2] was used to embed a wrap around mesh on it.

A large number of experiments were made with different values of $p$ and $n$. Some of these results are shown in Table 3. In calculating these efficiencies, we used the runtime of the serial algorithm on one processor, as the amount of work $W$.

From the results shown in Table 3, we can clearly see how the various mappings perform. The shuffling mapping does significantly better than either the checkerboarding or the modified checkerboarding mappings. For all schemes, the efficiencies increase with higher $n$, as the overheads due to communication and idling become a smaller fraction of the actual work. For SM, the efficiency goes all the way upto 1 (with increasing problem size), but for CM and MCM, it saturates at a smaller value than 1 as predicted by the analysis presented in Section 5.3 and 5.4. The saturation points for CM and MCM become smaller for larger number of processors as predicted by the analysis. Comparing the points where the efficiency saturates at Table 3 with the theoretical upper bounds in the efficiencies Tables 1 and 2 we see that they are very close. In particular, for $p = 16$, the predicted upper bound for CM and MCM are .17 and .53 respectively, which are quite close to the observed values .15 and .52. Similar statements are true for $p = 64$ and $p = 256$.

| $p \downarrow$ | $n \rightarrow$ | 120 | 200 | 280 | 360 | 440 | 520 | 600 | 2000 |
|---|---|---|---|---|---|---|---|---|---|
| | CM | .14 | .15 | .15 | .15 | .15 | .15 | .15 | .15 |
| 16 | MCM | .35 | .45 | .47 | .49 | .51 | .52 | .52 | .52 |
| | SM | .44 | .63 | .74 | .80 | .84 | .86 | .89 | .97 |
| | $n \rightarrow$ | 240 | 400 | 560 | 720 | 880 | 1040 | 1200 | 4000 |
| | CM | .10 | .11 | .11 | .11 | .11 | .11 | .11 | .11 |
| 64 | MCM | .28 | .34 | .37 | .39 | .41 | .42 | .42 | .42 |
| | SM | .38 | .57 | .69 | .75 | .80 | .82 | .86 | .96 |
| | $n \rightarrow$ | 480 | 800 | 1120 | 1440 | 1760 | 2080 | 2400 | 8000 |
| | CM | .08 | .09 | .09 | .09 | .09 | .09 | .09 | .09 |
| 256 | MCM | .22 | .28 | .31 | .33 | .37 | .38 | .38 | .38 |
| | SM | .31 | .50 | .64 | .69 | .75 | .78 | .83 | .95 |

Table 3: Efficiencies of the various mappings.

Finally, we constructed experimental isoefficiency graphs for the shuffling mapping. These graphs were obtained by performing a large number of experiments for a range of $n$ and $p$, and then collecting the points with equal efficiency. These graphs are shown in Figure 5, where the $x - axis$ shows the number of processors in $p^{1.5}$ scale and the $y - axis$ shows the work size $W$ in linear scale. Because the equal efficiency points are roughly on straight lines, we have that $W \sim p^{1.5}$, thus the isoefficiency of the shuffling mapping is $O(p^{1.5})$. The isoefficiency function can be also seen in Table 3. Entries in the same column for different values of $p$, are for problem sizes that increase at the same rate as $\sqrt{p}$. In the SM mapping, for $O(p \log p)$ isoefficiency, the efficiencies for different number of processors should be the same in each column. As we can see from this table, for small problems these efficiencies are not the same but decrease as the number of processors increases. This is because the constant of proportionality in the isoefficiency term is quite high. As the problem size increases though, the difference in the efficiencies decrease and for sufficiently large problems they are almost the same.

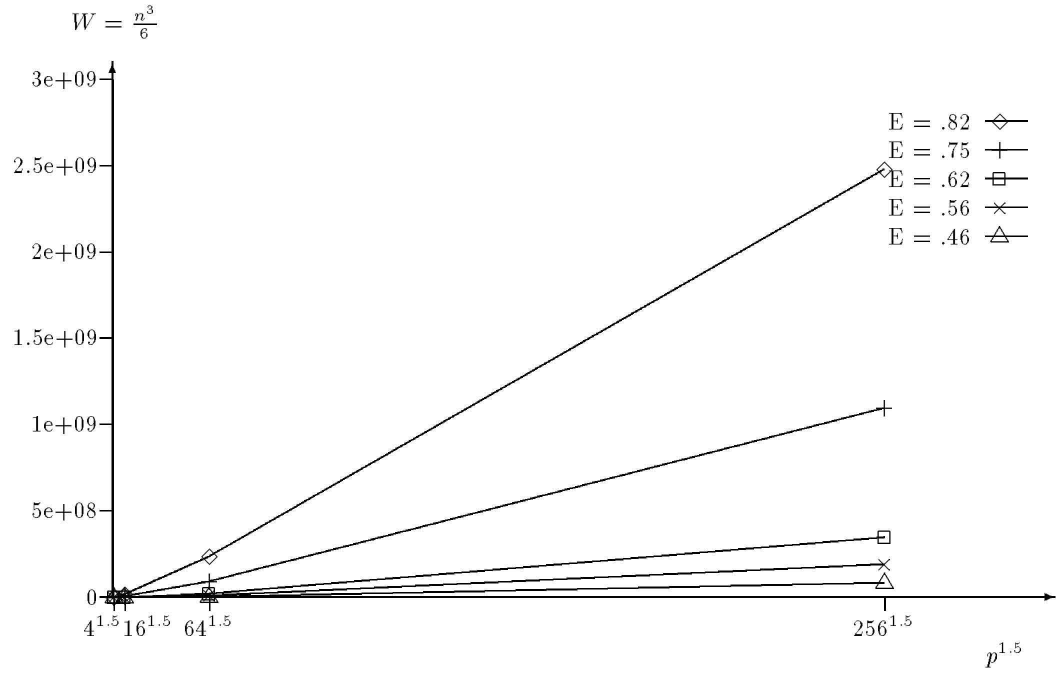


Figure 5: Isoefficiency graphs for the Shuffling Mapping

# 7 Conclusions

This paper presents a mapping of a two dimension systolic array on a mesh connected parallel computer that balances work among processors and minimizes communication costs for a class of systolic algorithms. In mapping of systolic algorithms similar to the parenthesization problem, it is particularly important that work be evenly distributed. For example, checkerboarding and modified checkerboarding mappings yield poor performance even if we assume that idling and communication time is zero. On the other hand shuffling evenly distributes the work among the mesh processors and yields efficiency approaching 1 for large enough problems. It can be shown that the shuffling mapping can be used to efficiently map a wide class of TSA algorithms onto mesh connected parallel computers. In particular, any TSA algorithm where the inputs to a cell are forwarded with no changes can be mapped efficiently onto a mesh parallel computer using this mapping.

# Appendix A Upper Bound on the Communication Performed by the Shuffling Mapping

Computing $T_{comm}^{P_{1,\sqrt{p}}}$ is more complicated than computing $T_{calc}^{P_{1,\sqrt{p}}}$ because the algorithm overlaps computations with communication. A precise analysis will complicate the presentation without further enhancing the qualitative and quantitative comparison of the different mappings. Instead we compute an upper bound on the amount of communication performed. In order for processor $P_{1,\sqrt{p}}$ to finish working on diagonal $k\sqrt{p}$ it has to receive the results from the previous $\sqrt{p}-1$ diagonals. The diagonals before $(k-1)\sqrt{p}$ are already known because processor $P_{1,\sqrt{p}}$ used them to compute diagonal $(k-1)\sqrt{p}$. The time spent receiving the $\sqrt{p}-1$ diagonals can be broken down to the time spent receiving the diagonal $(k-1)\sqrt{p}+1$ from processor $P_{1,1}$ and the time spent to receive the subsequent $\sqrt{p}-2$ diagonals. The former is: $((\frac{n}{\sqrt{p}}+1-k)t_w+t_s)(\sqrt{p}-1)$ and the later is: $(2(\frac{n}{\sqrt{p}}+1-k)t_c+(\frac{n}{\sqrt{p}}+1-k)t_w+t_s)(\sqrt{p}-2)$. Hence, an upper bound on $T_{comm}^{P_{1,\sqrt{p}}}$ is:

$$
\begin{aligned}
T_{comm}^{P_{1,\sqrt{p}}} &< \sum_{i=1}^{\frac{n}{\sqrt{p}}} \left( (\frac{n}{\sqrt{p}}+1-i)t_w+t_s \right) (\sqrt{p}-1) + \left( 2(\frac{n}{\sqrt{p}}+1-i)t_c + (\frac{n}{\sqrt{p}}+1-i)t_w+t_s \right) (\sqrt{p}-2) \\
&< \sum_{i=1}^{\frac{n}{\sqrt{p}}} \left( (\frac{n}{\sqrt{p}}+1-i)t_w+t_s \right) \sqrt{p} + \left( 2(\frac{n}{\sqrt{p}}+1-i)t_c + (\frac{n}{\sqrt{p}}+1-i)t_w+t_s \right) \sqrt{p} \\
&< (\frac{n^2}{\sqrt{p}}+n)t_w + (\frac{n^2}{\sqrt{p}}+n)t_c + 2nt_s
\end{aligned}
$$

This equation is an upper bound because while the diagonals needed by processor $P_{1,\sqrt{p}}$ are being transmitted, $P_{1,\sqrt{p}}$ will be performing computations thus, it will not be sitting idle waiting for results.

# Appendix B Tighter Efficiency Bound for Checkerboarding Mapping

In this section we are going to calculate a tighter upper bound on the efficiency of the checkerboarding mapping. This upper bound is obtained by considering the costs due to computation and idling. The overheads due to communication are ignored. For reasons, discussed in Section 5.3, we are going to concentrate on processor $P_{1,\sqrt{p}}$.

Let $T_1^i$ be the time from the point where processor $P_{1,i}$ for $1 \le i \le \sqrt{p}$, starts working on the first diagonal assigned to it until it computes the $\frac{n}{\sqrt{p}}^{th}$ diagonal. Note, that $T_1^i$ also includes some time spent on work done on the subsequent $\frac{n}{\sqrt{p}}$ diagonals.

Processor $P_{1,\sqrt{p}}$ starts working when it starts computing diagonal $n-2\frac{n}{\sqrt{p}}+1$. For that to happen, the $\frac{n}{2}-\frac{n}{\sqrt{p}}$ diagonal has to be computed. Note that this diagonal is the middle diagonal assigned to processor $P_{1,\frac{\sqrt{p}}{2}}$. We can compute the time required for this diagonal to be computed as the time required by processor $P_{1,\frac{\sqrt{p}}{2}}$ to start plus $T_1^{\frac{\sqrt{p}}{2}}$.

Hence, the time $T(n)$ required to compute diagonal $n$ at processor $P_{1,\sqrt{p}}$ is given by the recurrence relation:

$$T(n) = T(\frac{n}{2}-\frac{N}{\sqrt{p}}) + T_1^{\sqrt{p}} + T_2 \tag{10}$$

where $T_2$ is the time required by processor $P_{1,\sqrt{p}}$ to finish the last $\frac{n}{\sqrt{p}}$ diagonals. Also, we use $N$ to denote the problem size and differentiate it from the recurrence parameter $n$.

We can easily calculate the value of $T_1^i$, if we make a simple assumption. We assume that when a processors starts working on its first diagonal, it can also do some work on the other diagonals assigned to it. It can be easily seen that this holds for all the processors but the first two. Given this assumption, $T_1^i$ is:

$$\begin{aligned} T_1^i &= \frac{n^2}{p}(i\frac{n}{\sqrt{p}} - \frac{n}{\sqrt{p}}) - 2\sum_{k=1}^{\frac{n}{\sqrt{p}}}(\frac{n}{\sqrt{p}} - k)k \\ &\approx \frac{n^2}{p}(i\frac{n}{\sqrt{p}} - \frac{n}{\sqrt{p}}) - \frac{n^3}{3p\sqrt{p}} \end{aligned}$$

Similarly, $T_2$ can be easily computed and is:

$$T_2 = \frac{n^3}{3p\sqrt{p}}$$

Given these expressions, equation (10) becomes:

$$T(n) = T(\frac{n}{2} - \frac{N}{\sqrt{p}}) + \frac{N^2}{p}(\frac{n}{1} - \frac{N}{\sqrt{p}}) - \frac{N^3}{3p\sqrt{p}} + \frac{N^3}{3p\sqrt{p}}$$

Solving this equation with repeated substitution we get that:

$$\begin{aligned} T(n) &= \frac{n^2}{p}\sum_{i=0}^{\log\sqrt{p}-1}\left(\frac{n}{2^i} - \frac{n}{\sqrt{p}}\right) - \frac{n^3}{3p\sqrt{p}}(\log\sqrt{p} - 1) \\ &= \frac{n^3}{p}\left(\left(\sum_{i=0}^{\log\sqrt{p}-1}\frac{1}{2^i}\right) - \frac{log\sqrt{p}}{\sqrt{p}}\right) - \frac{n^3}{3p\sqrt{p}}(\log\sqrt{p} - 1) \end{aligned}$$

For sufficient large $p$, this equation simplifies to:

$$T(n) \approx 2\frac{n^3}{p}$$

Thus, the efficiency of the checkerboarding mapping is bounded by:

$$E \leq \frac{\frac{n^3}{6}}{2\frac{n^3}{p}p} = \frac{1}{12}$$

## Appendix C Tighter Efficiency Bound for Modified Checkerboarding Mapping

In this section we are going to calculate a tighter upper bound on the efficiency of the modified checkerboarding mapping. This upper bound is obtained by considering the costs due to computation and idling. The analysis presented here is an extension of the analysis presented in Appendix B.

In order to simplify the analysis, we can consider that modified checkerboarding treats the $\sqrt{p} \times \sqrt{p}$ mesh computer as a $2\sqrt{p} \times 2\sqrt{p}$ mesh. Because of this, $T(n)$ for modified checkerboarding is:

$$T(n) = \frac{n^3}{4p}\left(\left(\sum_{i=0}^{\log\sqrt{p}}\frac{1}{2^i}\right) - \frac{log2\sqrt{p}}{2\sqrt{p}}\right) - \frac{n^3}{24p\sqrt{p}}\log\sqrt{p}$$

Which for sufficiently large $p$ it simplifies to:

$$T(n) = \frac{n^3}{2p}$$

Thus, the efficiency of the checkerboarding mapping is bounded by:

$$E \leq \frac{\frac{n^3}{6}}{\frac{n^3}{2p}p} = \frac{1}{3}$$

## References


[1] A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation and Compiling, Vol. 1, Parsing.* Englewood Cliffs, NJ Prentice Hall, 1972.

[2] S. G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, 1989.

[3] R. Bellman and S. Dreyfus. *Applied Dynamic Programming.* Princeton University Press, Princeton, NJ, 1962.

[4] U. Bertele and F. Brioschi. *Nonserial Dynamic Programming.* Academic Press, NY, 1972.

[5] D. P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice-Hall, 1989.

[6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, McGraw-Hill, 1990.

[7] L. J. Guibas, H. T. Kung, and C. D. Thompson. *Direct VLSI Implementation of Combinatorial Algorithms.* In *Proceedings of Conference on Very Large Scale Integration, California Institute of Technology*, pages 509–525, 1979.

[8] Anshul Gupta and Vipin Kumar. *On the scalability of Matrix Multiplication Algorithms on Parallel Computers.* Technical Report TR 91-54, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, 1991.

[9] Oscar H. Ibara, Ting-Chuen Pong, and Stephen M. Sohn. *Parallel Recognition and Parsing on the Hypercube. IEEE Transactions on Computers*, 40(6):764–770, June 1991.

[10] Oscar H. Ibara and Stephen M. Sohn. *On Mapping Systolic Algorithms onto the Hypercube. IEEE Transaction on Parallel and Distributed Systems*, 1(1):48–63, January 1990.

[11] Guo jie Li and Benjamin W. Wah. *Parallel Processing of Serial Dynamic Programming Problems.* In *Proc. COMPSAC 85*, pages 81–89, 1985.

[12] Vipin Kumar and Anshul Gupta. *Analyzing Scalability of Parallel Algorithms and Architectures.* Technical report, TR-91-18, Computer Science Department, University of Minnesota, June 1991. A short version of the paper appears in the Proceedings of the 1991 International Conference on Supercomputing, Germany, and as an invited paper in the Proc. of 29th Annual Allerton Conference on Communuication, Control and Computing, Urbana,IL, October 1991.

[13] Vipin Kumar and V. Nageshwara Rao. *Parallel Depth-First Search, Part II: Analysis. International Journal of Parallel Programming*, 16 (6):501–519, 1987.

[14] H. Ney. *Dynamic programming as a technique for Pattern Recognition.* In *Proc. 6th Intl. Conf. Pattern Recognition*, pages 1119 – 1125, Oct. 1982.

[15] Wojciech Rytter. *On Efficienct Parallel Computations for Some Dynamic Programming Problems. Theoretical Computer Science*, 59:297–307, 1988.

[16] L. Valiant, S. Skyum, S. Berkowitz, and C. Rackoff. *Fast Parallel Computation of Polynomials Using Few Processors. SIAM J. Computing*, 12(4), 1983.

[17] Kumiko Wada and Nobuyuki Ichiyoshi. *A Distributed Shortest Path Algorithm and Its Mapping on the Multi-PSI.* In *Proceedings of Distributed Massively Concurent Computers.*, 1989.

[18] D. White. *Dynamic Programming.* Oliver and Boyd, Edinburgh, England, 1969.