10 Jul 1993

Parallel Algorithm Fundamentals and Analysis

Bruce M. McMillin
Missouri University of Science and Technology, ff@mst.edu

Hanan Lutfiyya

Grace Tsai

Jun-Lin Liu

Parallel Algorithm Fundamentals and Analysis CSC 93-05

BRUCE McMILLIN *[1], HANAN LUTFIYYA**[2], GRACE TSAI[1], and JUN-LIN LIU[1]

[1] *Department of Computer Science*
University of Missouri-Rolla
Rolla, MO 65401 USA
[2] *Department of Computer Science*
University of Western Ontario
London, ONTARIO N6A 5B7 CANADA

This paper appears, in its entirety, in the Preliminary Proceedings of the International Summer Institute on Parallel Computer Architectures, Languages, and Algorithms, July 5-10, 1993, Prague, Czech Republic. A revised version, which will appear in a volume published by the IEEE Computer Society Press, appears as Technical Report Number C.Sc. 93-17. available via anonymous ftp from "cs.umr.edu".

Key Words: Algorithm Design, Embeddings, Speedup, Class NC, Reasoning

* supported in part by the National Science Foundation under Grant Numbers MSS-9216479 and CDA-9222827, and, in part, from the Air Force Office of Scientific Research under contract number F49620-92-J-0546.

** supported in part by the National Sciences and Engineering Research Council of Canada (NSERC) under contract number OGP0138180-S365A2, and in part, from University of Western Ontario NSERC internal funding under contract number Z001A8-S365A1.

Abstract. This session explores, through the use of formal methods, the "intuition" used in creating a parallel algorithm design and realizing this design on distributed memory hardware. The algorithm class NC and the LSTM machine are used to show why some algorithms realize their promise of speedup better than others and the algorithm class NP is used to show why other algorithms will never be good for parallelization. Performance and correctness through cooperative axiomatic reasoning and temporal reasoning provide an additional basis for understanding parallel algorithm design and specification. Finally, the realities of algorithm design are presented through partitioning and mapping issues and models.

Table of Contents

1 Parallel Algorithms and Parallelization of Algorithms - Intuitive Design

Parallel processing can really only make sense if we understand how to program the parallel hardware that the technology is capable of producing. For example, 10,000 personal computers, each capable of 1 MFLOPS, has an enormous aggregate processing power of 10 GFLOPS, however, there is really no way to exploit this processing power for a realistic single job. Organizing these 10,000 PCs together, using a high-speed interconnection, such as in a *multicomputer*, helps, but the task remains to make the job run well. This is the study of parallel programming and parallel algorithms.

The goal of parallel programming and parallel algorithm study is to find a way to break a job into N units that can execute concurrently on N or fewer processors. Given the complexity of programming, in general, trying to program in parallel seems an insurmountable task. Indeed, a parallelizing compiler which transforms a sequential program into a parallel program would be very attractive. This idea, also coined the "Dusty Deck Syndrome" has received much research attention.

Parallelizing compilers work, for the most part, on identifying certain constructs within the sequential language. Execution profiles of computationally-intensive programs can show that often, only a few percent of code (by volume) accounts for 50% of the run time of the program. It's not hard to see where this lies. **DO** loops and computational kernels account for a great deal of a program's run time. Loops typically appear in program code as follows.

```
          DO i = 1, 100
100       a(i)=b(i)+c(i)
```

This loop parallelizes easily, and is easy for the compiler to detect and produce the following parallel (vector) code which executes all 100 assignments independently, in parallel.

```
a(1) = b(1) + c(1)
a(2) = b(2)+ c(2)
          ⋮
a(100) = b(100) + c(100)
```

or $a(1:100) = b(1:100) + c(1:100)$ Now, of course, not all loops are easily decomposed. Sometimes there are loop dependencies. These can be solved by the introduction of temporary storage. Other times, there are dependencies that cannot be removed, such as in the case of linear recurrences of the form $a_i = a_{i-1}b_i + c_i, i > 1$. FORTRAN code appears as follows,

```
          DO 100 i = 2, N
100       a(i) = a(i-1)*b(i) + c(i)
```

Notice that the data dependency between a(i) and a(i-1) cannot be parallelized completely. The (rather complex) solution

```
      a(1:N) = c(1:N)
      DO i = 1, log2 N
          DO in parallel for all Pj where 2^i ≤ j ≤ N
              a(j) = a(j) + b(j)*a(j-2^(i-1))
              b(j) = b(j)*(j-2^(i-1))
200       continue
```

builds up partial results in parallel, i.e. at i=2, at the end of the parallel statement, we have (for j $\geq$ 4):

a(j) = b(j)*b(j-1)*[b(j-2)*c(j-3)+c(j-2)]+b(j)*c(j-1)+c(j)

b(j) = b(j)*b(j-1)*b(j-2)

Now while the above construction is complex, once derived, a compiler can identify this looping structure and perform the appropriate parallel code substitution.

The problem with relying too much on compilers to do our work is twofold:

1. A compiler can only detect loop parallelism. Thus, there is still 50% of the run time unaccounted for that a compiler cannot easily detect. In terms of parallel program performance, this will limit the *Speedup* or increase in performance obtained by parallelism.
2. The sequential algorithm may actually obscure parallelism inherent in the problem such that even an ideal compiler can't extract it. Indeed, a sequential algorithm may not be the best parallel algorithm, at all.

In the next section we will examine the first issue, more closely, when we discuss the metrics of Speedup. The second issue is really one of language.

1.1 Language as an Impediment to Parallelism

The choice of language really can inhibit the expression of parallelism that may be inherent in an application. Consider the model of *Imperative Language Programming* which is the basis for FORTRAN, C, PASCAL, etc. An imperative language, consists of statements which are a sequence of predicate transformations on a program's state. For example, an imperative matrix multiplication $c_{l\times n} = a_{l\times m} b_{m\times n}$ is expressed as follows.

for i **from** 1 **to** l
  for j **from** 1 **to** m
    for k **from** 1 **to** n
      $c_{i,k} = c_{i,k} + a_{i,j} b_{j,k}$

This is the way that matrix multiplication is usually presented. However, it is not clear, at all, how to perform the operations in parallel. Certainly, since this is loop parallelism, we can create $l \cdot m \cdot n$ processes, as above. However, a better way is to re-examine the *specification* of matrix multiplication rather than its implementation in a particular (here imperative) language.

Matrix multiplication is, fundamentally, a collection of inner products of the elements of the multiplier and multiplicand matrices. This is expressed below, in a version of matrix multiplication expressed in FP [3].

Given a pair of matrices stored as a sequence of rows,
$< \mathbf{a}, \mathbf{b} >$, with $\mathbf{a} =< a_1, ..., a_l >$ and $a_i =< a_{i,1}, \ldots, a_{i,m} >$
$c \leftarrow$ *Inner_Product* $\cdot$ *Distribute_Left* $\cdot$ *Distribute_Right* $\cdot$ [$\mathbf{a}$, *transpose*($\mathbf{b}$)]
Whose evaluation results in:
$c \leftarrow$ *Inner_Product* $\cdot$ *Distribute_Left* $\cdot$ *Distribute_Right* $< \mathbf{a}, \mathbf{b}' >$
$c \leftarrow$ *Inner_Product* $\cdot$ *Distribute_Left* $<< a_1, \mathbf{b}' >, \ldots, < a_l, \mathbf{b}' >>$
$c \leftarrow$ *Inner_Product* $< p_1, p_2, \ldots, p_l >$ where $p_i =<< a_i, b'_1 >, \ldots, < a_i, b'_m >>$

By the Church-Rosser property, the Inner_Products may be applied in parallel in any order. Thus, we note that the execution order is neither constrained nor specified as in imperative languages. The maximum amount of parallelism is expressed by the functional program.

Now the FP example is rather extreme. No one is suggesting that everyone switch to functional languages simply to use parallel computing. Note, however, that by analyzing the specification of the problem, the observation that matrix multiplication is nothing more than a collection of inner products, yields not only the functional program above, but the imperative program, below.

do in parallel for $P_{ij}, i = 1, ..., l, j = 1, ..., m$
  for k **from** 1 **to** n
    $c_{i,k} = c_{i,k} + a_{i,j} b_{j,k}$

Thus, rather than express or constrain the computation of these inner products, as in the imperative algorithm, we just write an imperative program which is expressed in the fundamental parallel units of the problem. We then feed the inner product computations, in any order, to the processors of the system. Thus, rather than a parallel version of a sequential algorithm, this is a parallel algorithm.

Successful parallel programming consists of (1) specifying the problem, (2) identifying the fundamental units and their interaction, and (3) mapping these fundamental units to processes with their interactions specified by communication primitives.

Given that the only control we have in parallel programming, at the system level, is process creation and send/receive communication, all examples can be constructed using this primitive set of operations. Later we will present a more formal model of this in Hoare's CSP [17].

1.2 PARALLEL SORTING

Consider the problem of sorting an array **a** into ascending order using the a (very simple) Sequential Sorting Algorithm (Exchange Sort).

Sort N numbers a(1), a(2), ..., a(N) into ascending order
for i **from** 1 **to** N
    for j **from** 1 **to** N
        if (a(i) > a(j))
            temp=a(i)
            a(i)=a(j)
            a(j)=temp

This algorithm runs in N^2 comparisons. If we identify the fundamental units and operations in sorting, the compare/exchange is the basic function which operates on the array elements. If we have N processors available we should be able to make it run in N time by using the N processors to do N comparisons in parallel.

ODD-EVEN Transposition Sort If we arrange the N processors in a linear array and let processor P_i hold value a(i), then processors alternately exchange their values based on whether their index is even or odd.

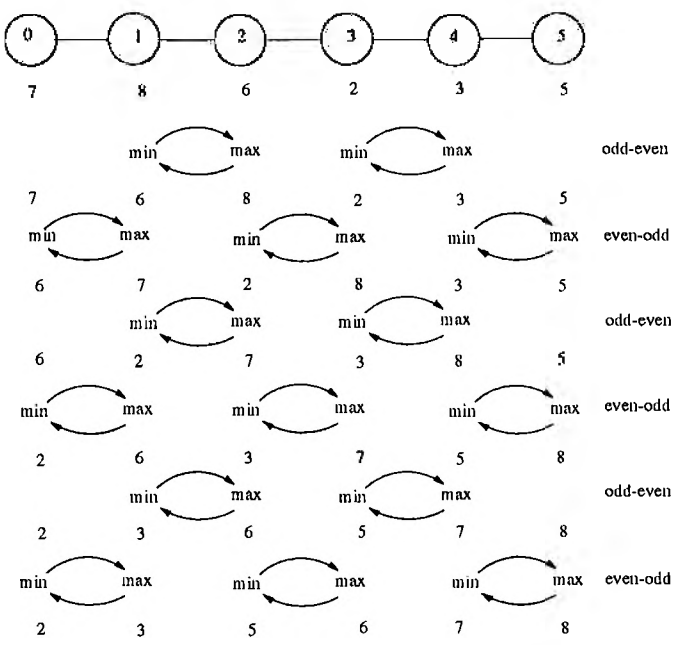


Fig. 1. Odd-Even Transposition Sort

Code for each processor P_i

for j = $0, N-1$

```
do in parallel for all P_i, i = 0, N - 1
    if j is even and i is even or j is odd and i is odd
            send a(i) to P_i - 1
            receive a(i) from P_i - 1
    else
            receive a(i+1) from P_i + 1
            if a(i+1) < a(i)
                temp=a(i)
                a(i)=a(i+1)
                a(i+1)=temp
            send a(i+1) to P_i + 1
end
```

This achieves the desired result, a parallel algorithm which runs in N time on N processors.

1.3 Relaxation

Perhaps the most important use of parallel computing is the relaxation methods for solving, iteratively, Partial Differential Equations of the form

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

A numerical approximation **U** to the solution **u** yields the matrix form

$$\mathbf{AU} = \mathbf{0}$$

where the matrix **A** is a sparse, tridiagonal, system of linear equations.

The problem of parallelizing a solution to this seems insurmountable. However, this problem is amenable to *Domain Decomposition* which splits the physical model's domain over the processors as in the point discretization of Figure 2.

Let $\mathbf{U} = (U_{i,j})$ be the approximation of the solution **u**

$$U_{i,j}^{(k+1)} = \frac{1}{4}(U_{i,j+1}^{(k)} + U_{i,j-1}^{(k)} + U_{i+1,j}^{(k)} + U_{i-1,j}^{(k)})$$

Each point (element) is iteratively solved as a function of its neighbors as in Figure 3.

1.4 NUMERICAL INTEGRATION

As another example of domain decomposition, consider the problem of an approximation to calculating π using numerical integration.

$$\pi \approx f(x) = 4\int_0^1 \frac{1}{1+x^2}dx$$

The natural numerical decomposition is to break the problem domain into strips and calculate the numeric function value at each strip to approximate the solution to the problem.

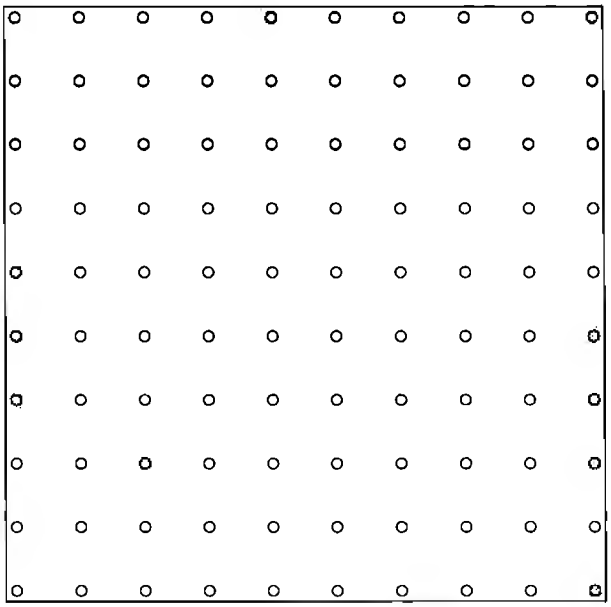


Fig. 2. Discretization of Physical Domain - Domain Decomposition

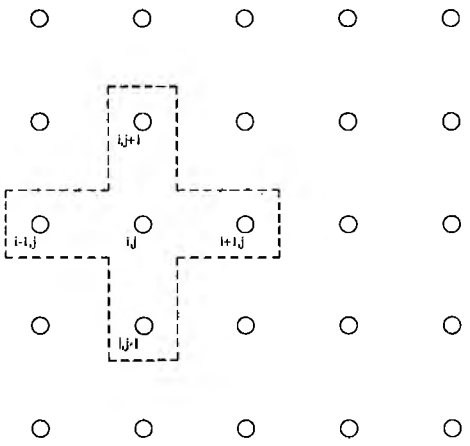


Fig. 3. Localized Computational Molecule

P_i:
return $\frac{1}{N} f(x_i)$

In parallel, each P_i gets $1/N$'th of the integration to perform, as in Figure 4. A tree reduction summation is used to sum up all the slices in logarithmic time.

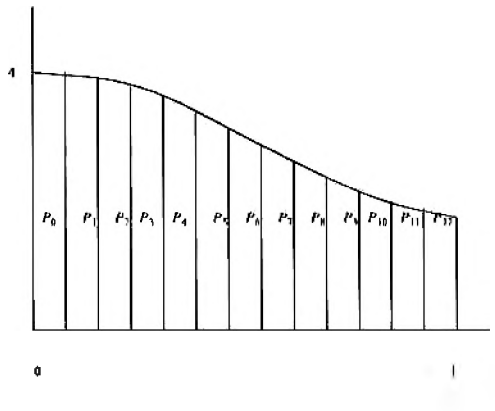


Fig. 4. Domain Decomposition

1.5 Summary

In creating a parallel algorithm, one must start with the specification of the problem to be solved. From this specification, identifiable units can be extracted that can be solved in parallel. Attempting to "engineer" a parallel solution from an existing sequential code, written in an imperative language, will not yield the best parallel algorithm since the imperative language imposes a computational order that does not always express the maximal parallelism present in the problem.

The remainder of this paper will explore metrics for measuring parallel performance, algorithmic classes of parallel algorithms, and a formal methods of reasoning about parallel programs.

2 Analysis of Parallel Algorithms

In the previous section, we presented a vague idea of how to measure the effectiveness of a parallel algorithm. In this section, we refine these concepts and present a theoretical basis for parallel algorithm performance.

2.1 Speedup

From a hardware standpoint, it's easy to build parallel hardware with enormous speed ratings. What the user desires is a machine to make his/her job run fast. If we assume that we can decompose the job into N parts, then *speedup* is just how much faster the decomposed job runs on N processors. Speedup measures address both the optimal and expected performance.

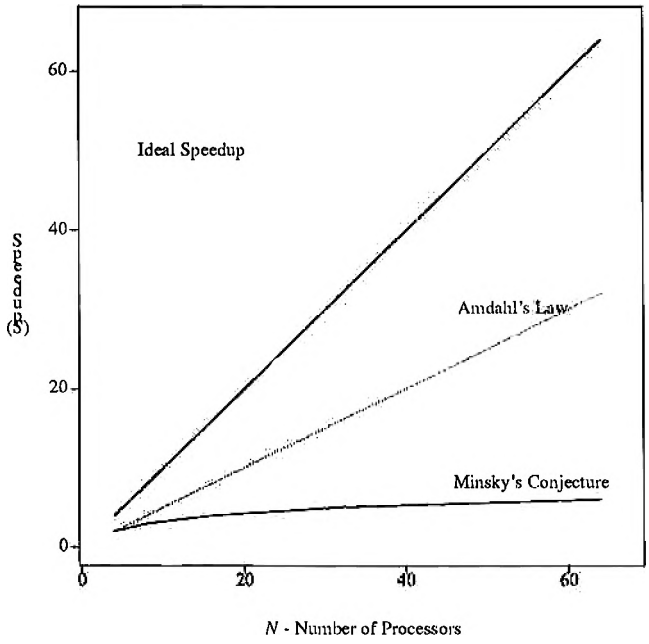


Fig. 5. Speedup Models

Figure 5 characterizes the best case, pessimistic case, and average case for possible speedups.

Minsky's conjecture [19] forms a lower bound on what we can reasonably expect from a parallel program. The key observation is that as N grows, the performance becomes dominated by system bottlenecks and communication. Thus, perhaps the best speedup, S is $O(log_2 N)$. This is a disappointing result, if true, as it says there is not much benefit from parallelism beyond only a few processors.

In sharp contrast to Minsky's conjecture is the notion of ideal speedup. For ideal speedup to be realized, the problem must be perfectly decomposed in N parts and no communication or system bottlenecks must occur. Then the speedup is linear, as N grows, the speedup $S = N$.

Between these two extremes, are two measures of what occurs when system bottlenecks, overhead, imperfect parallel decomposition occur.

Amdahl's law [1] treats every program as consisting of a sequential component s and a parallel component $p = 1 - s$. The crucial observation is that a program's speedup will be limited, severely, by the amount of non-parallelizable code. Simply put, if there are N processors, then the speedup S is bounded as follows:

$$S \leq \frac{s+p}{s+\frac{p}{N}}$$

For example, if $N = 1024$ and $s = 0$, then

$$S \leq \frac{1}{0+\frac{1}{1024}}$$

or $S \leq 1024$, which is, essentially, the ideal speedup case. However, if even a small sequential component is present, such as if $N = 1024$ and $s = 0.01$, then

$$S \leq \frac{1}{0.01+\frac{0.99}{1024}}$$

or $S \leq 91.18$.

Under the Amdahl's law speedup model, the limitations of parallelizing compilers become apparent. If we believe that 50% of the code is recognizable as parallel ($p = 0.5$), then 50% is not parallelizable ($s = 1 - p = 0.50$). Thus the maximum speedup is

$$\lim_{N\to\infty} S \leq \lim_{N\to\infty} \frac{1}{0.50+\frac{0.50}{N}}$$

or $S \leq 2$ no matter how many processors are used!

These results seem disappointing. However, [14] in 1988 observed that programs are made parallel, for the most part, as they are have run times which grow as the problem scales. This scaling can be a finer grid resolution or an increase in the number of time steps proportional to the number of processors in the system. However, the sequential time, which is the time to load the program, collect the results, and perform overhead calculations remains relatively constant over varying computational problem sizes. This *Scaled Size* model assumes that, by contrast to Amdahl's law, p is not independent of N. Thus, we can calculate a scaled speedup S_s, as

$$S_s = \frac{s+p\cdot N}{s+p}$$

Experimental results using this speedup measure report scaled speedups of 1020 on a 1024 processor machine [14]. There is still much debate, however, on the usefulness of this model.

2.2 Theoretical Basis for Speedup

Given the two speedup models for S and S_s given above, it is easy to calculate the speedup for a particular application. However, if the actual ratios p and s are not known, then experimentation is necessary. However, given that the best tools available are parallelizing compilers, determining p may be difficult since the p obtained is only an estimate of the amount of parallelism inherent in the problem. What is necessary is a way of classifying algorithms by their parallel complexity. The class $\mathcal{NC}$ is one such class. To explore the class $\mathcal{NC}$, we need to first examine the fundamental nature of parallel processes.

2.3 CSP

Hoare's model of concurrent programming, Communicating Sequential Processes (CSP) [17], is a model reflecting properties that should be in all concurrent programming languages. It was not intended to be used as a programming language per se, but it does reflect Hoare's concerns of proving the correctness of programs. However, CSP has provided a medium of discussion of synchronous systems and has inspired a great deal of development. One result is the multitasking and rendezvous properties of Ada. Hoare has suggested the following three properties that every concurrent language should have: the ability to express parallelism, communication primitives and non-determinism. This section provides an informal brief description of the syntax and meaning of CSP commands. Full details of CSP are contained in [17].

Communicating Sequential Processes (CSP) was proposed as a preliminary solution to the problem of defining a synchronous message-based language.

A CSP program consists of a static collection of processes. The basic command of CSP is $[\rho_1||...||\rho_n]$ expressing concurrent execution of sequential processes $\rho_1, ..., \rho_n$. Each individual process ρ_i has a distinct address space and consists of statements S_i. We can also express parallelism between program statements as well as between processes.

Coordination between processes is implemented by message exchange between pairs of processes. It involves the synchronized execution of *send*(output) and *receive*(input) operations by both processes. The send and receive operations in processes ρ_j and ρ_i take the following forms: $\rho_i!y$ *and* $\rho_j?x$, respectively.

Input command $\rho_j?x$ expresses a request to ρ_j to assign a value to the (local) variable x of ρ_i. Output command $\rho_i!y$ expresses a request to ρ_i to receive a value from ρ_j. Execution of $\rho_j?x$ and $\rho_i!y$ is synchronized and results in assigning the value of y to x. $\rho_j?x$ and $\rho_i!y$ are said to be a *matching pair* of communication statements. We define a *communication sequence* of process ρ_i as the sequence of all communications that ρ_i has so far participated in.

The alteration command allows for a path to be non-deterministically chosen from a set of paths. The repetition rule allows for repeated non-deterministic choosing of a path from a set of paths.

The alteration and repetition commands are as follows:

$$\textbf{if } b_1; c_1 \rightarrow S_1 \,\square ... \square\, b_n; c_n \rightarrow S_n \textbf{ fi}$$

$$\mathbf{do}\, b_1; c_1 \rightarrow S_1 \Box ... \Box b_n; c_n \rightarrow S_n\, \mathbf{od}$$

Alteration and repetition are formed from sets of guarded commands. A guarded command b; c → S consists of a guard b;c and a command S. In the guard, b is a boolean expression and c is either skip or one of the communication primitives. The symbol ";" is used as a delimiter for separating different program statements. If b is false, the guard is failed. If b is true and c=skip, the guard is ready. If b is true and c is one of the communication primitives, then the guard is prepared to communicate with the process named in the communication primitive. It is ready when the other process is prepared to communicate and blocked at other times.

Execution of an alteration command selects a guarded command with a ready guard and executes the sequence c;S. If c is skip, execution is independent of other processes. If c is a communication command, then a matching communication command must be executed simultaneously. When some guards are blocked and none are ready, the process is blocked and must wait. If all guards are failed, the process aborts.

Execution of the repetitive command is the same except that, whereas execution of alternation selects one guarded command and is completed, for repetition the selection is repeated until all guards are failed, at which time execution of the repetition is repeated until all guards are failed, at which time execution of the repetition is completed.

2.4 Complexity

The questions of complexity and computability that exist for sequential computer programs, are also interesting questions for concurrent/parallel computer programs. If the Turing Machine is the abstract computational model for a sequential program, what is the corresponding model for a concurrent program and how does this model relate to the sequential Turing Machine model?

From [18], the fundamental measures of complexity are *parallel time*, *space*, and *sequential time*. If we have an abstract model which provides these three measures, then we can succinctly define speedup and characterize classes of algorithms which are amenable to parallelism.

If the model of concurrent computation is represented by CSP, concurrent programs are really expressed by sequential programs that communicate with each other. Since the Turing Machine is the model of sequential programs, it is natural to express a concurrent program as a set of communicating Turing machines. Specifically, a concurrent program is represented by a Multitape Turing machine which has a read-only input tape, k work tapes ($k > 1$), and a write-only output tape. Roughly, the input tape and output tape correspond to the message passing that occurs in CSP ?,! barrier rendezvous and each work tape corresponds to the internal storage of one of the k processes of the CSP program.

Definition 1. Formally, a Turing Machine (TM) is described by

$$M = (Q, I, \Sigma, \delta, \mathrm{b}, q_0, F)$$

where Q is the finite set of states, Σ is the tape alphabet, $I \subseteq \Sigma$ is the input, δ is the move function, b is a special blank symbol, $q_0 \in Q$ is the start state, and $F \subset Q$ is the set of final states.

Definition 2. For a TM M and input w, $t(w)$ is the total number of steps taken for input w and

$$t(n) = max\{t(w) \mid |w| \leq n\}$$

is the time complexity of M.

Definition 3. For a TM M and input w, $s(w)$ is the total maximum length of any work tape used for input w and

$$s(n) = max\{s(w) \mid |w| \leq n\}$$

is the space consumption of of M.

Definition 4. Let ID be the instantaneous description of M,

$$ID \equiv \Sigma^* Q \Sigma^* \equiv xqy$$

where xy are tape contents and the tape head is scanning the leftmost symbol of y in state q and $\vdash$ represents a move of M.

Definition 5. Let $ID_0 \vdash ID_1 \vdash ID_2 \vdash \cdots$ be a computation of M for input w. If, in two successive steps, $ID \vdash ID' \vdash ID''$ a work tape moves in different directions, we say a head changes its movement during $ID \vdash ID' \vdash ID''$. Define $(i, j), i < j$ as a *phase* of this computation if no work tape head changes its movement direction during $ID_i \vdash ID_i + 1 \vdash ID_i + 2 \vdash \cdots \vdash ID_j$ where in $ID \vdash ID'$, every tape head moves $R, L,$ *or* S where R and L are different directions and S is no movement.

Next we define a machine which will help relate the phases to the concept of data dependencies between sequential processes through message passing.

Definition 6. Let a *Transform Machine* be a TM constructed from M adding a special state q'. Upon entering q', it removes all the contents from the input tape, copies the output tape to the input tape, changes the work tape and output tape to blanks, and works normally starting in state q_0.

Definition 7. The *width complexity* $w(n)$ is the maximum total length of the input and output tape contents during the computation for all input of length $\leq n$.

Remark. What these definitions show is that if we can use n work tapes in a single phase, independently, this implies there are no data dependencies between the work tapes. The end of a phase (entering state q') implies that a communication or synchronization is necessary. Thus, in the Turing Machine formulation, the width complexity corresponds to the total amount computational space (complexity) and the space complexity corresponds to the longest space complexity of an individual work tape (process).

A special type of transform machine is of interest, since it describes a computation which is amenable to parallelism in logarithmic time.

Definition 8. If a transform machine satisfies

$$s(n) = O(log(w(n)),$$

it is a *Log-Space Transform Machine* (LSTM).

For example, there is a LSTM that satisfies the computation of a tree-reduction summation.

Example 1. An LSTM which satisfies the computation of $\sum x_i$ for $x_1\#x_2\#\cdots\#x_k$ where the $X_k's$ are binary numbers as input as follows.

In Phase 1, M gets $y_1\#y_2\#y_3\#\cdots$ on its output tape where $y_i = x_{2i} + x_{2i+1}$.

In Phase 2, M $y_1\#y_2\#\cdots$ becomes the input and M gets $z_1\#z_2\#\cdots$ on its output tape where $z_i = z_{2i} + z_{2i+1}$.

This continues until the output is $\sum x_i$. This clearly takes $log\ k$ phases. The width complexity $w(n) = O(n), k \leq n$ and the phase complexity is $log\ k$.

The problems that can be solved by a LSTM form a complexity class, $\mathcal{NC}$.

Definition 9. A problem is in $\mathcal{NC}$ if there exists an LSTM solving it in polynomially related phase $O(log^* n)$ and width $O(n^*)$ where $g(n) = f^*(n)$ if $g(n) = p(f(n))$ for some polynomial, p.

Thus, the class $\mathcal{NC}$ represents the class of nicely parallelizable problems with time polynomial in the logarithm of the size of the problem (poly-log) using only a polynomial number of processors. Clearly any problem in $\mathcal{P}$ is in $\mathcal{NC}$, since any problem in $\mathcal{NC}$ when solved serially, is in $\mathcal{P}$. However, the reverse is not necessarily true since, for example, the best-known parallel algorithm for maximum flow is $O(n^2 log n)$ steps using $O(n)$ processors.

2.5 $\mathcal{NP}$-Completeness and Parallel Computing

While the results above show that the class $\mathcal{NC}$ contains problems amenable to parallel computing, there are algorithm classes in which parallel computing is ineffective.

The class of $\mathcal{NP}$-Complete problems, or those solvable in nondeterministic polynomial time form just such a class. Since it is not known if any $\mathcal{NP}$ problems can be solved in deterministic polynomial time, attempting to solve an $\mathcal{NP}$-complete problem requires exponential time on a sequential computer.

Since, by the above discussion, that our notion of a parallel computer is really expressed by a multitape Turing Machine, and, since multitape and single tape Turing Machine computations are related, then, by the Church-Turing hypothesis, any $\mathcal{NP}$-Complete problem can be expressed as a parallel algorithm on the multitape Turing machine. However, by our notion of speedup, S, using

N processors, the best speedup is N, a linear factor. However, an exponential problem, E) grows in some exponential power of N, $E = O(c^N)$. Thus, since a parallel machine grows in power, only linearly, it cannot effectively reduce the exponential complexity of the problem. Put more succinctly, parallel computers only reduce the complexity of an exponential problem by a polynomial factor S, thus, leaving the complexity exponential since $E/S = C^N/N$ is still exponential.

However, parallel computers are useful in evaluating expensive hueristics for approximation to the solution of $\mathcal{NP}$-Complete problems. Techniques such as simulated annealing [27] provide good results, but are computationally complex. Parallel computing can help speed their evaluation.

3 Interconnection Networks and Embeddings

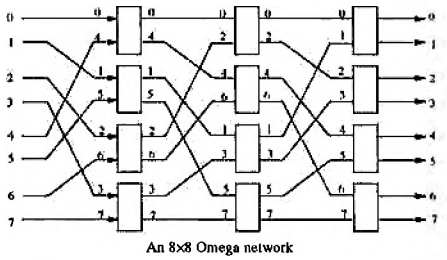
In the presentation so far, we have assumed that all processors are connected to each other (a completely connected network). The crossbar switch [19] attempts to connect each processor to each other processor. However, the number of switch elements grows as the square of the number of processors, making this technology infeasible for large multicomputer networks. The bus interconnection [19], by contrast, is inexpensive, but exhibits a performance bottleneck as interprocessor communication grows.

Multistage interconnection networks attempt to minimize the cost of interconnecting processors by providing a subset of possible interconnection patterns between the processors, at any one time. Examples of multistage interconnection networks are shown in Figure 6. Each network is arranged in n stages where each stage has N/k $k \times k$ switches, each with $N = k^n$ ports. Thus, each processor can communicate with each other processor using n hops in the switch, however, as mentioned above, only a subset of simultaneous connections are possible.

The multistage interconnect is the basis for many commercial and research parallel processors such as PASM [38] and the IBM RS/6000-based POWER-PARALLEL System[20]. However, if we examine the examples of Section 1 the communication patterns between processors are all *nearest neighbor*. Indeed, the most natural parallel algorithms result from domain decomposition into spatially local communication patterns such as mesh, ring, or tree. Thus, a fixed architecture which can be a host to these guest graphs is all that is really necessary.

A fixed interconnection topology is the usual choice in constructing multicomputers. The topology is based on a graph theoretical model in which processors are represented by nodes or vertices and links are represented by edges so that all links are bidirectional.

A *path* is a sequence of links from the source node to a destination node. The *path length* (distance) between two nodes is the minimum number of links between these two nodes. The *degree* of a node is the number of links (bidirectional) connecting to a node.



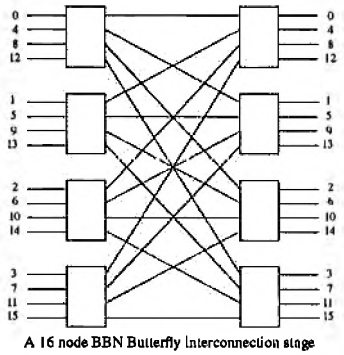


Fig. 6. Sample Multistage Interconnection Networks

3.1 Graph Embedding

The need for the embedding arises from at least two different directions. First, with the widespread availability of distributed memory architectures based on the hypercube interconnection scheme, there is an ever-growing interest in the portability of algorithms developed for architectures based on other topologies, such as linear arrays, rings, two-dimensional meshes, and complete binary trees, into the hypercube. Clearly, this question of portability reduces to one of embedding the above interconnection schemes into the hypercube. Second, the problem of mapping parallel algorithms onto parallel architectures naturally gives rise to graph embedding problems. Graph embedding problems have applications in a wide variety of computational situations. For example, the flow of information in a parallel algorithm defines a program graph and embedding this into a network tell us how to organize the computation on the network. Other problems that can be formulated as graph embedding problems are laying out circuits on chips, representing data structures in computing memory, and finding efficient program control structures.

The problem of mapping a graph representing the computation and communication needs of the program onto the underlying physical interconnection of a multiprocessor so as to minimize the communication overhead and maximize

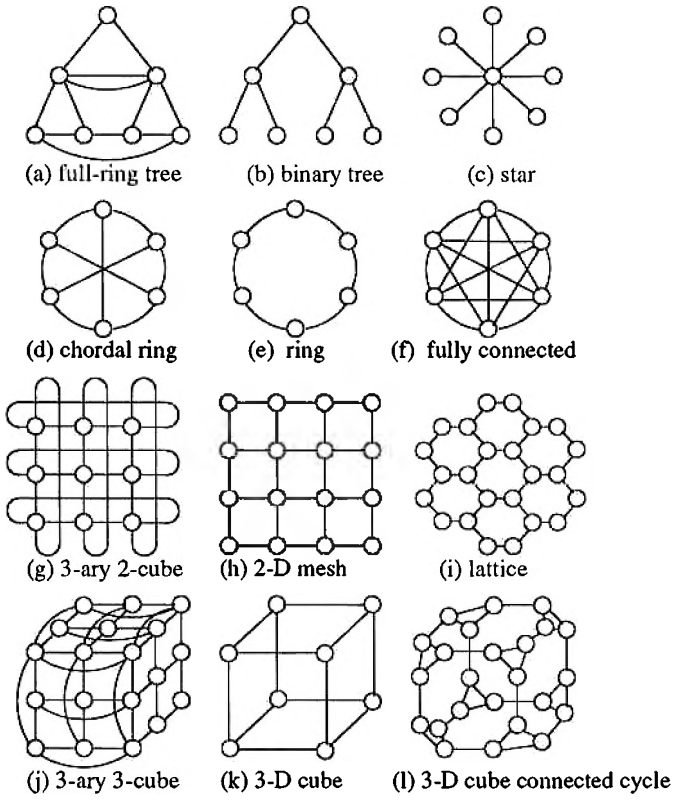


Fig. 7. Some Interconnection Topologies

the parallelism is called the mapping problem. The mapping problem is the assignment of processes to processors so as to maximize the number of pairs of communicating processes that fall on pairs of directly connected processors.

In mapping problems, the *guest graph* G is the network topology that we are interested in simulating using a *host graph* H. Let V_G and V_H denote the vertex sets of the graph G and H, respectively, and E_G and E_H denote the edge sets of the graph G and H, respectively. An *embedding* f of a graph G into a graph H is a mapping of the vertices of G into the vertices of H, together with a mapping of the edges of G into the simple paths of H such that if $e = (u, v) \in E_G$, then $f(e)$ is a simple path of H with endpoints $f(u)$ and $f(v)$. If $f(e)$ has length greater than one, then it has one or more intermediate nodes which are all nodes on the path other than the two endpoints. An embedding f is *isomorphic* if it is injective and for each $(u, v) \in E_G, (f(u), f(v)) \in E_H$. Throughout this paper, unless indicated otherwise the term "embeddings" will always means isomorphic embeddings, and the terms "embedding" and "mapping" will mean the same

and used interchangeably.

It has been known for a long time that the general graph embedding problem (i.e., subgraph isomorphism problem) is NP-complete. It was shown that the embedding of general graphs into the binary hypercube is also NP-complete [8]. However, with rich interconnection structure the hypercube contains as a subgraph many the regular structures (i.e., rings, two-dimensional meshes, higher-dimensional meshes, and almost complete binary trees). Most of the mapping research in these years has dealt with effectively simulating these regular structures in the hypercubes, (for example, [40]).

Let f be an embedding function which maps a guest graph G into a host graph H. $|V_G|$ denotes the cardinality of the set V_G. Terminology related to the mapping problem are formally defined as follows.

Definition 10. The *expansion* of the mapping is the ratio of the size (in number of nodes) of the host graph to that of the guest graph, that is, $E_f = \frac{|V_H|}{|V_G|}$ If the embedding is injective, then the expansion is a measure of processor utilization.

Definition 11. The *edge dilation* of edge $(i,j) \in E_G$ is $dist(f(i),f(j))$. The *dilation* of the mapping is $D_f = max(dist(f(i),f(j)), \forall (i,j) \in E_G$. The *average edge dilation* is $\frac{1}{|E_G|} \sum_{(i,j) \in E_G} dist(f(i),f(j))$. The dilation of a mapping represents the communication delay between the communication nodes.

Definition 12. The *congestion of an edge* $e' \in E_H$ is the cardinality of $e \in E(G)$: e' is in path $f(e)$. That is, $\sum_{e \in E_G} |e' \cap E_{f(e)}|$. The *congestion* of the mapping is $max\{\sum_{e \in E_G} |e' \cap E_{f(e)}|\}$, $\forall e' \in E_H$. The *average congestion* of the mapping is similarly defined.

Definition 13. The *max-load* is the maximum number of nodes in G that are mapped to a node in H. Max-load = 1 if the mapping is one-to-one.

It should be noted that unit dilation implies unit congestion. Thus the class of dilation-1 embeddable graphs in a hypercube is a proper set of the class of congestion-1 embeddable graphs. If each node of the guest graph is mapping to a distinct node of the host, the slow down due to nearest neighbor communication in the original graph being extended to communication along paths is a function of the length of the path (i.e., edge dilation) and the congestion of the edges on the path.

3.2 The k-ary n-cube Interconnection Topology

One of the most general type of interconnection network is the k-ary n-cube which has k^n nodes organized as a cube with dimension n and k nodes in each dimension. Each node i is identified by an n-digit radix k number, the b-th digit of the number represents the node's position in the b-th dimension. The nodes are interconnected to their nearest neighbors in a radix k representation as follows.

Definition 14. If $i_{n-1}\cdots i_0$ is the radix k representation for node i, then its neighbors in the interconnection are

$$i_{n-1}i_{n-2}\cdots i_{b+1}i_b^+ i_{b-1}\cdots i_0$$

and

$$i_{n-1}i_{n-2}\cdots i_{b+1}i_b^- i_{b-1}\cdots i_0 for\ each 0 \le b \le n-1$$

where

$$i_b^+ = (i_b + 1) mod\, k$$

and

$$i_b^- = (i_b - 1) mod\, k$$

An example of a 3-ary 2-cube is shown in Figure 7 g.

Some special cases of this topology are the $k = 2$ case of the hypercube or boolean n-cube. For $n = 2$ a superset of a a k dimensional mesh is generated and $n = 1$ specifies a ring.

Boolean n-cube Various supercomputer architectures interconnecting hundreds or thousands of processors have been proposed for many years. The Hypercube is used on both SIMD and MIMD parallel processors. Some commercial examples are the NCUBE/2, the Intel iPSC/860, and the CM-2.

An n-cube system has $N{=}2^n$ nodes (processors) indexed from 0 to $2^n - 1$ and there is a link between any two nodes if and only if the binary representations of their indices differ by exactly one bit. An n-cube can be recursively constructed by combining two $(n-1)$-cubes. Let $(a_{n-2}\ldots a_0)$ be an index in $(n-1)$-cube. Then in n-cube, there is a link between two corresponding nodes in $(n-1)$-cube, $(0a_{n-2}\ldots a_0)$ and $(1a_{n-2}\ldots a_0)$. A 2-ary 3-cube is shown in Figure 8.

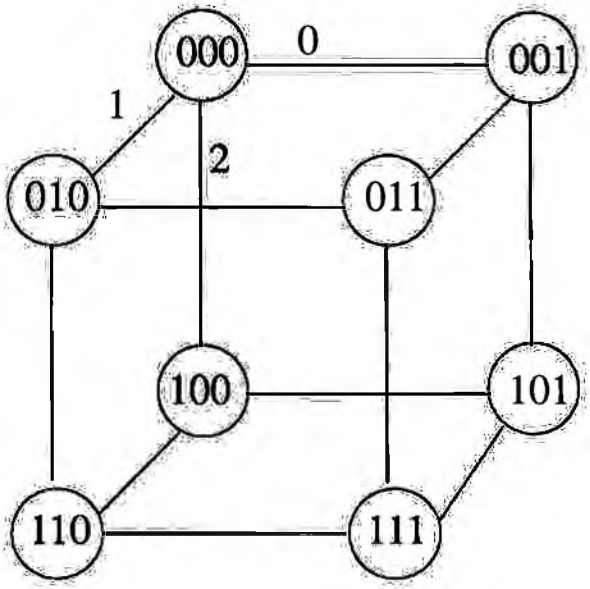
3.3 Pattern Embedding in a Hypercube

The hypercube is a powerful topology because it is a superset of many other topologies, such as ring, mesh, and tree. Commonly, each of these nodes in these topologies is given a binary representation. However, the binary representation chosen needs to preserve the nearest neighbor adjacencies present in the k-ary n-cube representation. Fortunately, the Gray-code provides just such a representation.

Definition 15. A Binary Reflected Gray Code (BRGC) G_k is a code of length k such that

$G_{k-1}(i_l)$ is the $k-1$-bit Gray code representation of digit i_l of the radix $k-1$ number i and $G_{k-1}(i_l)^R$ is its reversal.

$$G_k = \begin{cases} \{0,1\} & \text{if } k = 1 \\ \{0G_{k-1}(0), 0G_{k-1}(1), \ldots, 0G_{k-1}(2^{k-1}-1), \\ 1G_{k-1}(2^{k-1}-1), 1G_{k-1}(2^{k-1}-2), \ldots, 1G_{k-1}(0)\} \\ = \{0G_{k-1}, 1G_{k-1}^R\} & k > 1 \end{cases}$$



epsfxsize=2.1in

Fig. 8. A 2-ary 3-cube

Ring Embedding Rings are of interest, and are of increasing interest, due to the computational problems that arize in genetics. One of the central questions of molecular biology is the discovery of the semantics of DNA. Just knowing the syntax, that is, the sequence, tells the biologist little. The biologist must understand the biochemical functions of the DNA. To understand the semantics, one needs to know the relationship between DNA and proteins. The essence of the problem is that given a set of protein sequences, efficient alignment-matching algorithms are needed that can deal elegantly with insertion, deletion, substitution, and even gaps in the series of sequence elements. One way of measuring the optimality of an alignment is by computing a score based on a matrix of weights reflecting the similarity between pairs of sequences. In some situations a penalty is subtracted for each gap introduced. Such a score can be computed by a dynamic programming algorithm in time proportional to the product of the lengths of the sequences.

The subsequence matching problem can be formulated as follows:

Given two sequences A, B, of symbols chosen from a same domain

$$A = (a_1, a_2, ..., a_n),\ B = (b_1, b_2, ..., b_m),$$

find the subsequences

$$A' = (a_{i_1}, a_{i_2}, ..., a_{i_x}),\ B' = (b_{j_1}, b_{j_2}, ..., b_{j_x})$$

$$where\ 1 \leq i_1 < i_2 < ... < i_x \leq n,\ 1 \leq j_1 < j_2 < ... < j_x \leq m$$

which maximizes the comparison function $C(A', B')$. C can depend on the symbols a_{i_l}, b_{j_k} in A' and B' and on the numbers of symbols in A and B which are omitted between successive symbols in A' and B' (gaps).

For such comparison functions, one can use a dynamic programming algorithm to determine the best subsequence match for a given pair of sequences A, B in serial time $O(mn)$ where n and m are the length of the sequences A and B. This dynamic programming algorithm can best be understood by considering the matrix

$$C_{r,s} = max \begin{cases} 0 \\ C_{r-1,s-1} + D(a_r, b_s) \\ C_{r-1,s} + g \\ C_{r,s-1} + g \end{cases}$$

where the gap constant $g < 0$, and D is a correlation function between single elements [24].

A parallel version of the dynamic programming algorithm is quite straightforward to derive [10]. Since computing the value of $C_{r,s}$ only depends on knowing the values of $C_{r-1,s}$, $C_{r,s-1}$, and $C_{r-1,s-1}$, we see that all of the elements on one anti-diagonal of the matrix can be computed simultaneously if the values along the two previous anti-diagonals are known. That is, for a fixed value of t, the matrix elements $C_{t-s,s}$ can be computed simultaneously for all s provided that they are known for $t-1$ and $t-2$. Thus, one can parallelize the above algorithm by computing successive anti-diagonals of the matrix $C_{r,s}$ on successive time steps. This is represented schematically in Figure 9. The algorithm requires $n+m-1$ time steps and m processors to compare proteins of length m and n.

Since each communication in the above algorithm is nearest neighbor, mapping the ring computational structure to directly connected processors is important.

Theorem 16. *A k-ary 1-cube is a subgraph of a 2-ary n-cube when $n = log_2 k$ and $k = 2^j$ for some integer j.*

Proof. The idea is to number the nodes of the k-ary 1-cube using a BRGC. For each node i of the k-ary 1-cube, re-number that node by $G_k(i) = g_{k-1}g_{k-2}\cdots g_l \cdots g_0$. The predecessor and successor nodes of the k-ary 1-cube are numbered (from Definition 14 with $n = 1$)

$$i^- \text{ and } i^+$$

where

$$i^+ = (i+1) \bmod k \text{ and } i^- = (i-1) \bmod k$$

which, using the definition of G_k are the nodes

$$g_{k-1}g_{k-2}\cdots\overline{g}_l\cdots g_0$$

and

$$g_{k-1}g_{k-2}\cdots\overline{g}_{l+1}g_l\cdots g_0$$

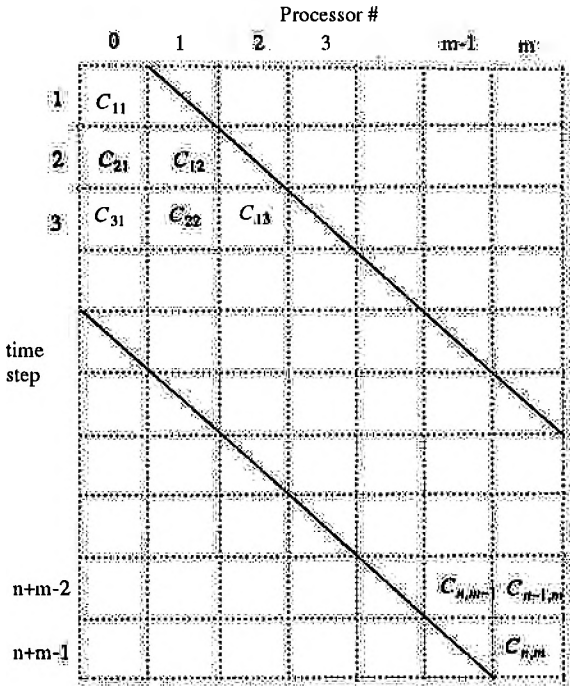


Fig. 9. Diagram indicating activity of processor i at time step t. If $1 \leq t-p \leq n$, then processor i computes $C_{t-p,p+1}$ at step t. Otherwise, the processor is inactive.

Corollary 17. *A ring of length of* 2^m *can be mapped into the 2-ary* n*-cube when* $2 \leq m \leq n$.

Proof. Since a 2-ary $n-1$ cube is a subgraph of a 2-ary n-cube, the result is immediate.

If we notice that a ring of length 2^n exists within a G_n because a path of length of 2^{n-1} exists within the first half of $G_n(= 0G_{n-1})$ and is connected to a path of length of 2^{n-1} within the second half of $G_n(= 1G^R_{n-1})$, then we can also construct rings of any even length by starting with shorter paths.

Corollary 18. *A ring of length* $p = 2q$ *can be mapped into the 2-ary* n*-cube when* $4 \leq p \leq 2^n$.

Proof. Find a path of length q as follows

$$\{0G_{n-1}(i), 0G_{n-1}(i+1), \ldots, 0G_{n-1}(i+q-1),$$
$$1G_{n-1}(i+q-1), 1G_{n-1}(i+q-2), \ldots, 1G_{n-1}(i)\}$$

For example, of a ring of length

$$p = 12 : \{0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011\}$$

Mesh Embedding Of great interest in Computational Science and Engineering is programs whose structure is the mesh. Consider the mode fluids problem [36] of cavity-driven flow whose physical domain chosen is shown in Figure 10. The pair of non-linear coupled differential equations 1,2 that describe this flow are easily solved sequentially using a standard second-order central differencing scheme. Central differencing calculates the new values at a particular point by taking a weighted average of the values of the nearest neighbors, as shown in Figure 11, where the weights are dependent on the flow patterns.

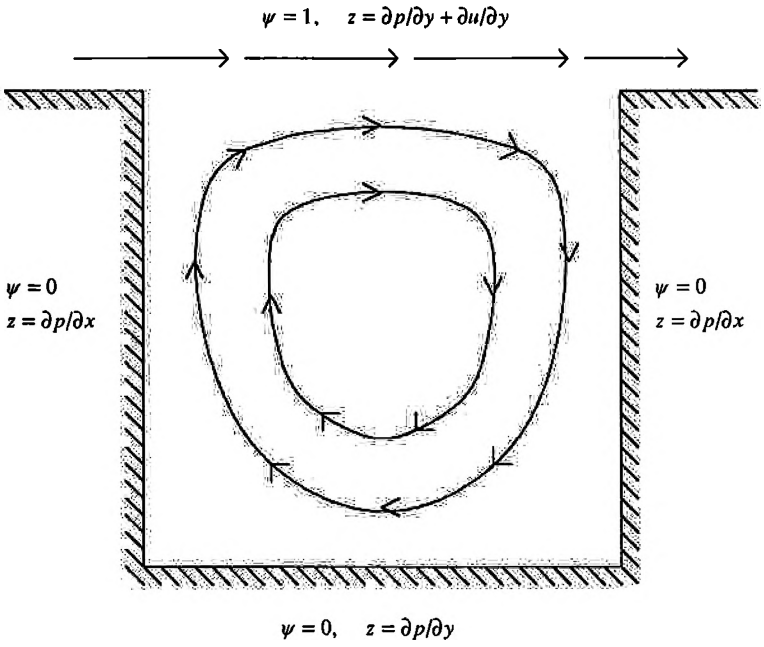


Fig. 10. Cavity Driven Flow

$$\zeta = -\nabla^2 \psi \tag{1}$$

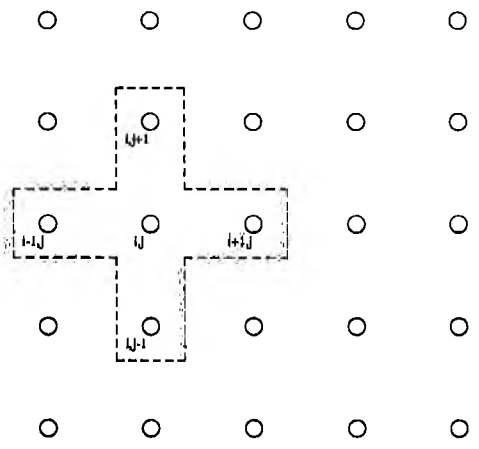


Fig. 11. Localized Computational Molecule

$$\frac{\partial \zeta}{\partial t} + \frac{\partial}{\partial x}(u\zeta) + \frac{\partial}{\partial y}(v\zeta) = \frac{1}{Re}\nabla^2\zeta \tag{2}$$

where $u = \frac{\partial \psi}{\partial y}$ and $v = -\frac{\partial \psi}{\partial x}$.

These two equations represent the flow conditions in the physical domain. Lines of constant stream function, ψ, value are parallel to the local flow, while the vorticity, ζ, is a measure of the local shearing rate, or swirl, in the flow.

These equations were solved using successive over-relaxation with the resulting discrete equations as follows:

$$\psi_{i,j}^{k+1} = \psi_{i,j}^{k} + \frac{\omega}{2(1+\beta^2)}\left[\psi_{i+1,j}^{k} + \psi_{i-1,j}^{k} + \beta^2\left(\psi_{i,j+1}^{k} + \psi_{i,j-1}^{k}\right) - \zeta_{i,j}\Delta x^2\right] \tag{3}$$

$$\zeta_{i,j}^{n+1} = \zeta_{i,j}^{n} + \Delta t\left[\frac{-u_{i+1,j}^{n}\zeta_{i+1,j}^{n} - u_{i-1,j}^{n}\zeta_{i-1,j}^{n}}{2\Delta x} + \frac{-v_{i,j+1}^{n}\zeta_{i,j+1}^{n} - v_{i,j-1}^{n}\zeta_{i,j-1}^{n}}{2\Delta y}\right.$$

$$\left. + \frac{1}{Re}\left(\frac{\zeta_{i+1,j}^{n} + \zeta_{i-1,j}^{n} - 2\zeta_{i,j}^{n}}{\Delta x^2} + \frac{\zeta_{i,j+1}^{n} + \zeta_{i,j-1}^{n} - 2\zeta_{i,j}^{n}}{\Delta y^2}\right)\right] \tag{4}$$

where ω is the over-relaxation factor and $\beta = \frac{\Delta x}{\Delta y}$.

Superscript k indicates the current iteration value and n is the value at the current time. The boundary values for ζ are calculated by using first-order accurate, away-from-the-wall equations:

$$\zeta_{i,w} = -\frac{2}{\Delta y^2}\left(\psi_{i,w} - \psi_{i,w+1}\right)\left[+\frac{u_{i,w}}{\Delta y}\right] \tag{5}$$

$$\zeta_{w,j} = -\frac{2}{\Delta x^2}(\psi_{w,j} - \psi_{w+1,j}) \quad (6)$$

In equations 5 and 6, w is the location of the boundary, and the bracketed term is only used at the top of the cavity, where the external flow affects the values.

The standard solution method is to take an initial guess of the values of u, v, and ζ, along with a Δt appropriate for the fineness of the grid, and iterate equation 4 once. These values are then used to iterate equation 3 to convergence, update the values of u and v, calculate the boundary values for ζ, then repeat the process until the values of ζ and ψ have both met desired convergence criteria.

Optimal Matrix Multiplication (in the abstract sense) As another mesh problem, consider Gentleman's Algorithm [13] which is an explicit parallel solution using a 2D mesh of processors to multiply two matrices.

Assume we have N^2 processors arranged in an $N \times N$ mesh. Each processor $\rho_{i,j}$ holds $a_{i,j}$ and $b_{i,j}$ and we have a toroidal mesh (an easily implemented subgraph of an n-cube).

Optimal $OMEGA(n)$ Algorithm
foreach $\rho_{i,j}$ SEND and RECEIVE to
    left circular shift all $a'_{i,j}s$ by $i-1$
    up circular shift all $b'_{i,j}s$ by $j-1$
foreach $\rho_{i,j}$
    $c_{i,j} \leftarrow a_{i,j}b_{i,j}$
    do n-1 times
        left circular shift $a_{i,j}$;up circular shift $b_{i,j}$
        $c_{i,j} \leftarrow c_{i,j} + a_{i,j}b_{i,j}$

Example 2. Consider the example of matrix multiplication shown in Figure 12. The result $c_{2,3}$ is calculated as follows, $c_{2,3} \leftarrow a_{2,1}b_{1,3}+a_{2,2}b_{2,3}+a_{2,3}b_{3,3}+a_{2,0}b_{0,3}$

Embedding Results for Meshes

Theorem 19. *A k-ary 2-cube is a subgraph of a 2-ary n-cube when $n = 2\log_2 k$ and $k = 2^j$ for some integer j.*

Proof. As in the proof of Theorem 16, we number the digits of the k-ary graph using G_j. Specifically, for each node $i = i_1 i_0$ of the k-ary 2-cube, re-number that node by $G_j(i_1)G_j(i_0)$. Consider the 4 neighbors of i,

$$\begin{matrix} i_1 i_0^+ & i_1 i_0^- \\ i_1^+ i_0 & i_1^- i_0 \end{matrix}$$

where

$$i^+ = (i+1) mod\, j$$

and

$$i^- = (i-1) mod\, j$$

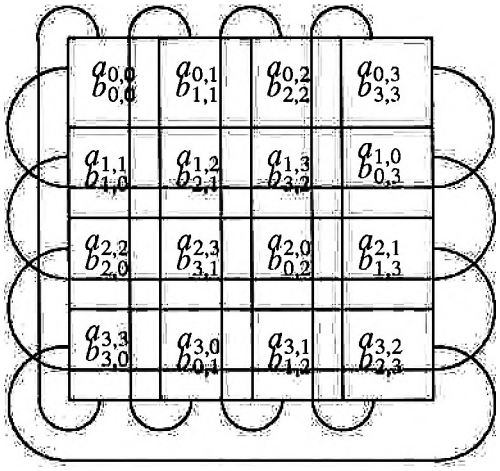


Fig. 12. Toroidal Shift

and their Gray code ordering

$$G(i_1)G(i_0^+)\ G(i_1)G(i_0^-)$$
$$G(i_1^+)G(i_0)\ G(i_1^-)G(i_0)$$

Since we change only one dimension of i at a time for each neighbor, we can consider each mapping individually, as in the ring case. Using the definition of G_j, a particular i_m, $G(i_m)$'s neighbors are the nodes

$$g_{j-1}g_{j-2}\cdots\overline{g}_l\cdots g_0$$

and

$$g_{j-1}g_{j-2}\cdots\overline{g}_{l+1}g_l\cdots g_0$$

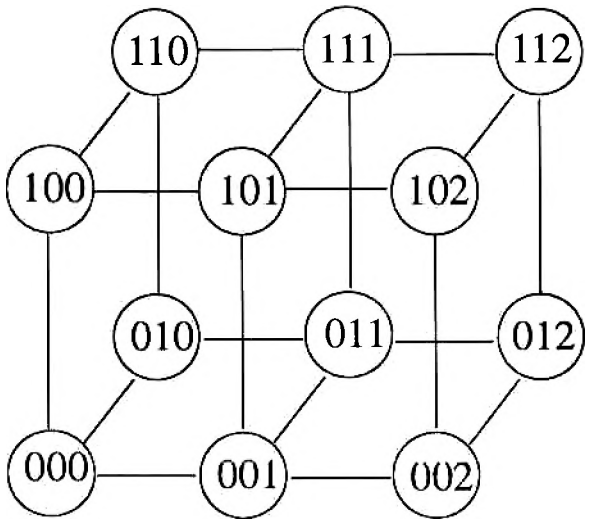
Thus, each $G_j(i_m)$ enumerates a 2-ary j-cube. Taking the cross product of $G_j(i_1) \times G_j(i_0)$ yields a 2-ary n-cube.

A d-dimensional mesh is an $m_0 \times m_2 \times ...m_{d-1}$ mesh in the d dimensional space. An example is shown in Figure 13

Corollary 20. *An $m_0 \times m_1 \times \cdots \times m_{d-1}$ mesh in d-dimensional space, where $m_i = 2^{k_i}$ and $\sum_{i=0}^{d-1} k_i = n$ can be mapped into a 2-ary n-cube where the mapping is $G_{k_{d-1}}(i_{d-1}) \times \cdots \times G_{k_1}(i_1) \times G_{k_0}(i_0)$.*

Tree and Pyramid Embedding The final topology to be considered is the tree. Tree computations occur more infrequently than either the mesh or ring, however, an extension of the tree, the pyramid, occurs frequently in multigrid algorithms.

The first, rather surprising, result is an impossibility proof on tree embedding.



epsfxsize=2.1in

Fig. 13. 3D Mesh Interconnection

Definition 21. T_n denotes a complete binary tree of $2^n - 1$ nodes.

Theorem 22. *A binary tree T_n with $2^n - 1$ nodes cannot be embedded into an n-cube for $n \geq 3$.*

Proof. It is enough to show that by adding an additional node and all relevant edges to T_n that we cannot reconstruct a 2-ary n-cube (of 2^n nodes).

Observe that for the leaf nodes of the tree, there are no cross edges such as in Figure 14a, since that would form an odd-length cycle, which, in a 2-ary n-cube, cannot exist. Similarly, there can be no edges, as in Figure 14b.

Given these constraints, we can add edges to the tree at each leaf node. Since $n - 1$ neighbors of each leaf need to be found, a total of $(n-1)2^{n-1}$ edges and one node need to be added. Since whether the tree has an even or an odd depth influences the assignment of edges, there are two cases. Notationally, we put the root at level 1 of the tree.

Case 1 If n is odd, then the number of nodes which can receive edges from the leaf nodes is

$$\sum_{i=1}^{\frac{n-1}{2}} 2^{2i-1}$$

which is simply a count of the nodes at levels at an odd distance from the leaves of the tree. Each of these nodes already has 3 edges, so they absorb

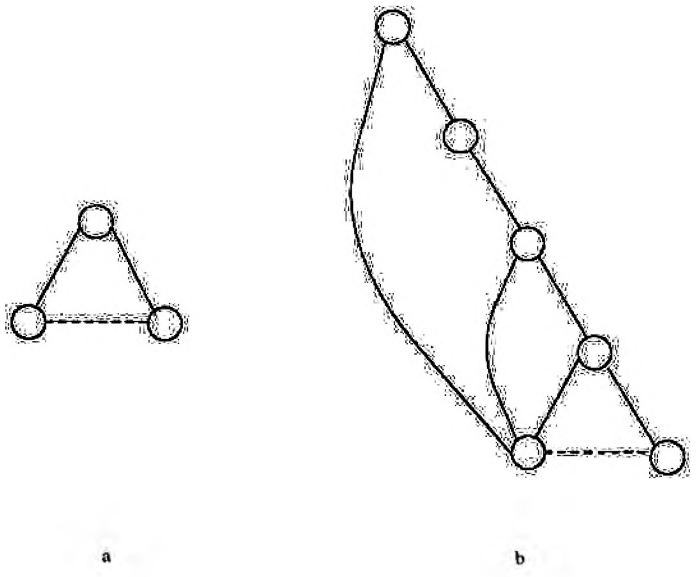


Fig. 14. There can be no cross edges (a), nor edges between the leaf nodes and a node at a level an even height away from the leaf

$n-3$ more edges (since each node in a 2-ary n-cube has a total of n edges). The extra node can absorb n edges. Thus,

$$(n-3)\sum_{i=1}^{\frac{n-1}{2}} 2^{2i-1} + n < (n-1)2^{n-1} \; for \; n \geq 3$$

Thus, since the inequality is strict, there are not enough vertices to absorb all the edges necessary to reconstruct the 2-ary n-cube.

Case 2 If n is even, then the number of nodes which can receive edges from the leaf nodes is

$$\sum_{i=0}^{\frac{n-2}{2}} 2^{2i}$$

Here each node has 3 edges except for the root which has 2. The extra node can absorb n edges. Thus,

$$(n-3)\sum_{i=0}^{\frac{n-2}{2}} 2^{2i} + n < (n-1)2^{n-1} \; for \; n \geq 3$$

Again, since the inequality is strict, there are not enough vertices to absorb all the edges necessary to reconstruct the 2-ary n-cube.

We can, however, embed T_{n-1} in a 2-ary n-cube (left as an exercise), but the node utilization is poor,

$$E_f = \frac{2^n}{2^{n-1}-1} \approx 2$$

which is about a 50% waste of nodes.

We can do better if we allow for a dilation, $D_f = 2$.

Theorem 23. *An n-cube contains a binary tree of height n with ($2^n - 1$ nodes) with a dilation $D_f = 2$.*

Proof. Define S_n as the graph obtained by taking 2 disjoint T_{n-1} and connecting their roots by a path of length 3 as in the construction of S_4 shown in Figure 15

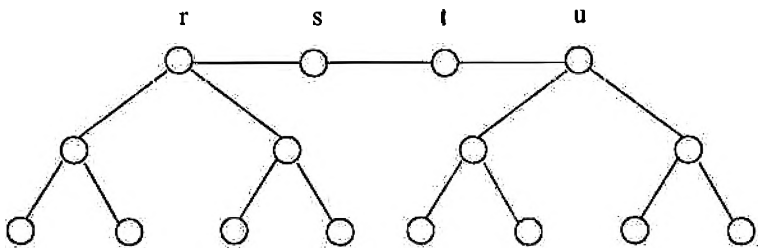


Fig. 15. Tree S_4

Now, we need to show that for $n \geq 3$, roots r and u of T_{n-1} can be labeled such that $dist(r, u) = 3$.

By induction, assume that S_{n-1} is a subgraph of a 2-ary $n-1$-cube with

$$G_{n-1}(r) = 001\cdots11$$
$$G_{n-1}(s) = 011\cdots11$$
$$G_{n-1}(t) = 111\cdots11$$
$$G_{n-1}(u) = 111\cdots10$$

Now find two disjoint subgraphs S'_{n-1} and S''_{n-1} in a 2-ary $n-cube$. In the construction, S'_{n-1} is obtained by prefixing every label of S_{n-1} with a 0. Thus,

$$0G_{n-1}(r') = 0001\cdots11$$
$$0G_{n-1}(s') = 0011\cdots11$$
$$0G_{n-1}(t') = 0111\cdots11$$
$$0G_{n-1}(u') = 0111\cdots10$$

and for every node i in S_{n-1} $G_{n-1}(i) = g_{n-2}g_{n-3}\cdots g_0$ relabel i in S''_{n-1} by $1G^R_{n-1}(i) = 1g_0g_1\cdots g_{n-2}$. Thus,

$$1G_{n-1}(r'') = 11\cdots100$$

$$1G_{n-1}(s'') = 11\cdots110$$
$$1G_{n-1}(t'') = 11\cdots111$$
$$1G_{n-1}(u'') = 101\cdots11$$

These two trees are depicted in Figure 16.

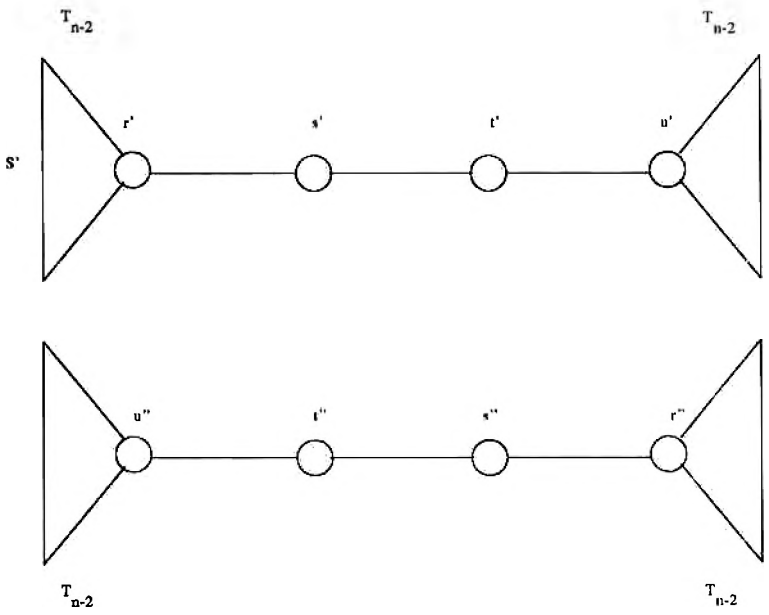


Fig. 16. Two subtrees created by relabeling their node numbers

We can now form $S_n = S''_{n-1} \cup S'''_{n-1} + \{(s'u''), (t't''), (u's'')\} - \{(t'u'), (t''u'')\}$ yielding the construction of Figure 17.

An example is shown in Figure 18.

While trees, themselves, do not hold a great deal of interest, practically, an extension of the tree, the pyramid, is computationally interesting.

The Multigrid Method The initial idea behind multi-grid is that convergence time decreases dramatically with an improved initial guess. From this idea, it seems reasonable to use a coarse grid to get a rough solution, and then interpolate

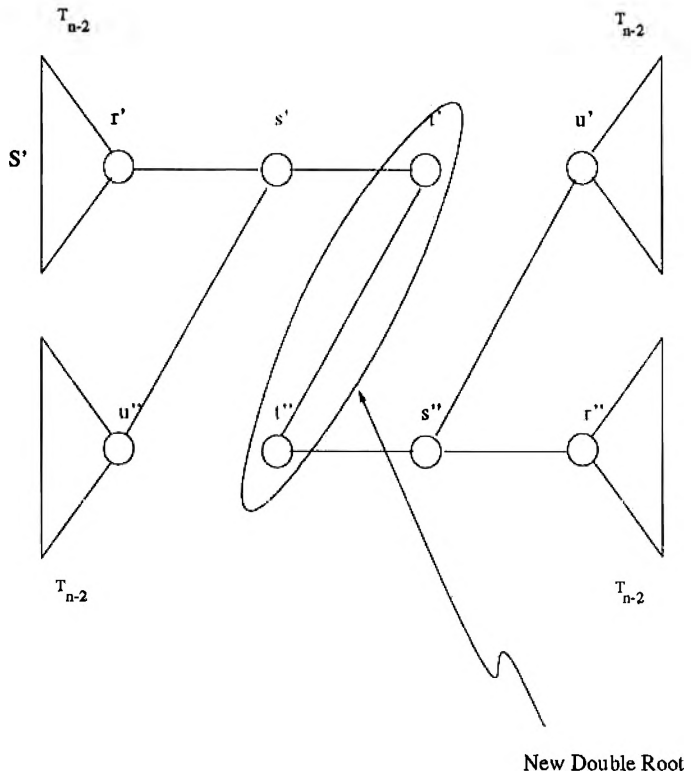


Fig. 17. Tree Constructed from S' and S''. The new root node is represented by s', t', and t''.

this answer to finer and finer arrays as shown in Figure 19. Although this does work, multi-grid methods are much more powerful than this simple concept. Given the system of equations

$$\mathbf{AU} = \mathbf{F}, \tag{7}$$

the usual procedure is to guess a solution, $\mathbf{V}$, to $\mathbf{U}$, then calculate $\mathbf{AV}$ and correct the guess by comparison to $\mathbf{F}$. The estimate $\mathbf{V}$ is known to be some amount $\mathbf{E}$ away from the exact solution, giving

$$\mathbf{U} = \mathbf{V} + \mathbf{E}$$

and by substituting into equation 7,

$$\mathbf{A}(\mathbf{V} + \mathbf{E}) = \mathbf{F}.$$

Initially, this doesn't help since neither $\mathbf{U}$ nor $\mathbf{E}$ is known. However, after rearranging,

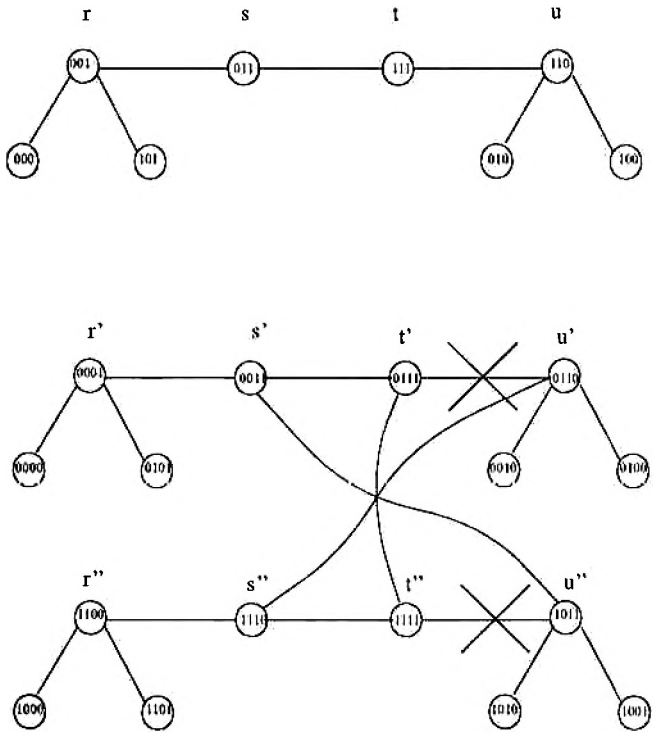


Fig. 18. An Example of Tree Embedding with $D_f = 2$

$$\mathbf{AE} = \mathbf{F} - \mathbf{AV}$$

and finally,

$$\mathbf{AE} = \mathbf{R}, \tag{8}$$

where **R** denotes the residual, **R** = **F** - **AV**. This resulting equation can be solved exactly as the first equation, since all of the variables except **E** are known.

The reason why equation 8 is solved instead of equation-7 has to do with the size and frequency of the error. If the error in the value is small, but not yet small enough to satisfy convergence criteria, and the absolute value of the result is large, the small error will be hard to distinguish from the result. If instead, the values are subtracted out, the magnitude of the error will then be centered around zero, so the relative size of the error will be magnified.

The observed frequency of the error is dependent on the coarseness of the array, as shown in Figure 20. What may be seen as a relatively smooth change at the finest level appears as rapid changes when restricted to a coarser level.

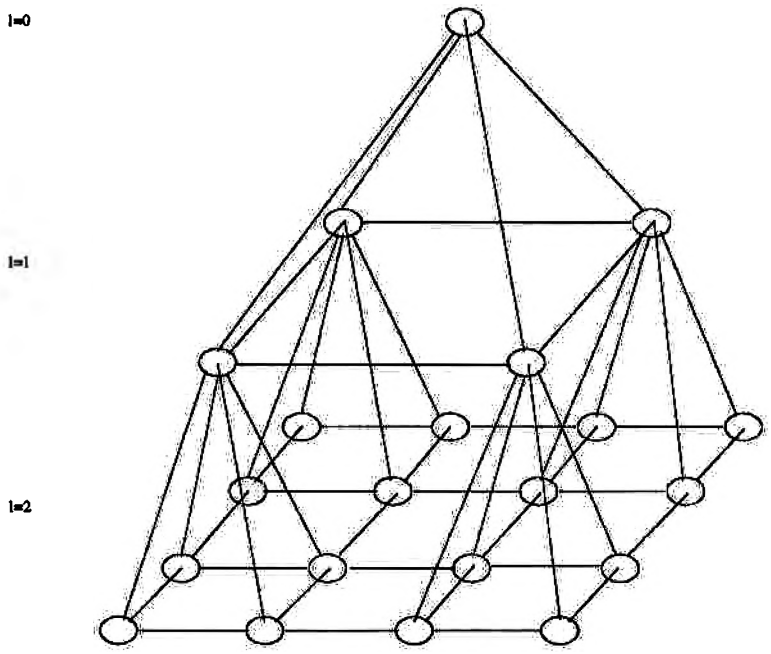


Fig. 19. Multgrid structure for N=16 Processors at the Finest Level

Thus, solving the errors at a coarser level increases the speedup of the solution by damping out the errors faster, along with increasing convergence rate due to better guesses.

As illustrated in Figure 21 and as described by [4], there are many ways to implement the multi-grid idea. In the figure, level 0 represents the finest array of points, while level 3 is the coarsest.

In the V-cycle, level 0 does a set number of iterations of equation 7, then passes its residuals to level 1. Level 1 then iterates equation 8 and passes its residuals to level 2, where the process is repeated until the coarsest level is reached. When the coarsest level finishes its computations, it passes the error corrections back down through the levels, until level 0 is reached.

The W-cycle takes additional advantage of the speed of the coarser grids by having them also do some improvement of the errors before the errors get passed back down the levels. This helps speed up the damping out of the smooth changes since the coarser levels converge faster.

Finally, the full multi-grid (FMV) cycle takes advantage of both the error correction and improved initial guesses. Instead of starting at the finest level,

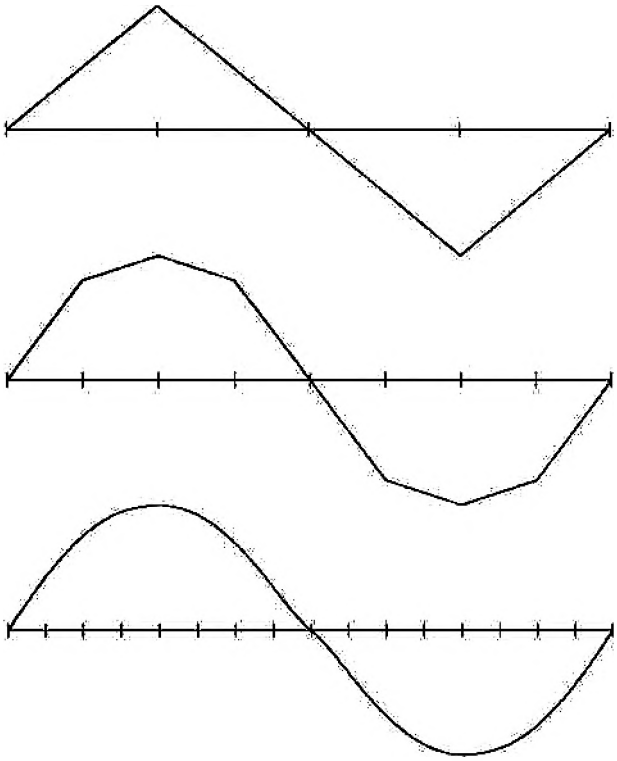


Fig. 20. Error Frequency Reduction Using Multgrid

FMV-cycles start at the coarsest arrays and compute an initial guess that is passed down to the next level. That level then does a few iterations and does a single V-cycle to improve its guesses before passing them down. Once the lowest level is reached, the process continues as a regular V-cycle.

Embedding of Pyramid into n-cube

In observing Figure 19, it is clear that embedding the pyramid into the n-cube is not going to be possible with $D_f = 1$ since between each pair of levels of the pyramid, there are odd length cycles. However, $D_f = 2$ mappings exist. The mapping makes use of the following Gray code.

Definition 24. A Hierarchical Binary Reflected Gray Code (HBRGC) is a BRGC such that

$$h(G_n(i), G_n(i + 2^j)) = 2 \; when \; i + 2^j \leq 2^n - 1, j > 0$$

Definition 25. The Hierarchial Binary Reflected Gray Code HG_k is a code of length k such that

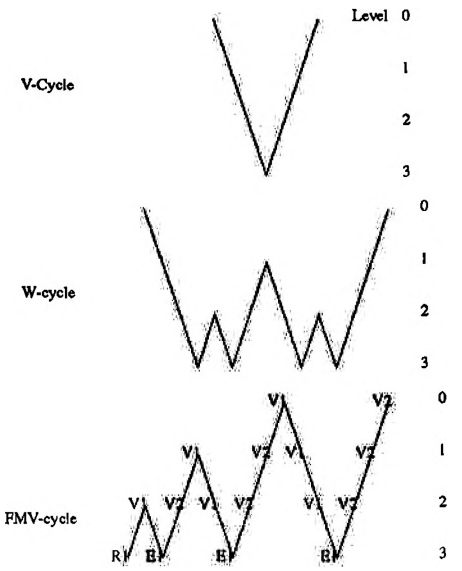


Fig. 21. V and W Cycles of the Multigrid Method

$$HG_k = \begin{cases} \{0,1\} & \text{if } k = 1 \\ \{HG_{k-1}(0)0, HG_{k-1}(0)1, HG_{k-1}(1)1, HG_{k-1}(1)0, \ldots, \\ \quad HG_{k-1}(2^{k-1}-2)0, HG_{k-1}(2^{k-1}-2)1, \\ \quad\quad HG_{k-1}(2^{k-1}-1)1, HG_{k-1}(2^{k-1}-1)0\} & k > 1 \end{cases}$$

If we define $R_l(HG_k) = \{HG_{k-1}(0)10^{l-1}, HG_{k-1}(1)10^{l-1}, \ldots, HG_{k-1}(2^{k-1}-2)10^{l-1}, HG_{k-1}(2^{k-1}-1)10^{l-1}\}$, which is just HG_k1, then R_kHG_k defines level $k+1$ of a two-dimensional pyramid. Level k of the pyramid is created by $R_k + 1(HG_k - R_k(HG_k))$ which yields HG_k0, or the subset of HG_k whose nodes are at least a power of 2 distance away from the nodes of $R_k(HG_k)$. The process recurses until the entire pyramid is constructed. In general, at level $l+1$, of the pyramid, each node at that level is labeled $HG_{k-l}(i)10^l$, thus reflecting that each node at level $l+1$ is, at most, a distance of 2 away from child nodes at level $l+2$.

Example 3. HG_2 generates the following pyramid depicted in Figure 22.

- $HG_2 = \{000, 001, 011, 010, 110, 111, 101, 100\}$
- $R_2(HG_2) = \{001, 011, 111, 101\}$
- $R_1(HG_2 - R_2(HG_2)) = \{010, 110\}$
- $R_0(R_1((HG_2 - R_2(HG_2)))) = \{100\}$

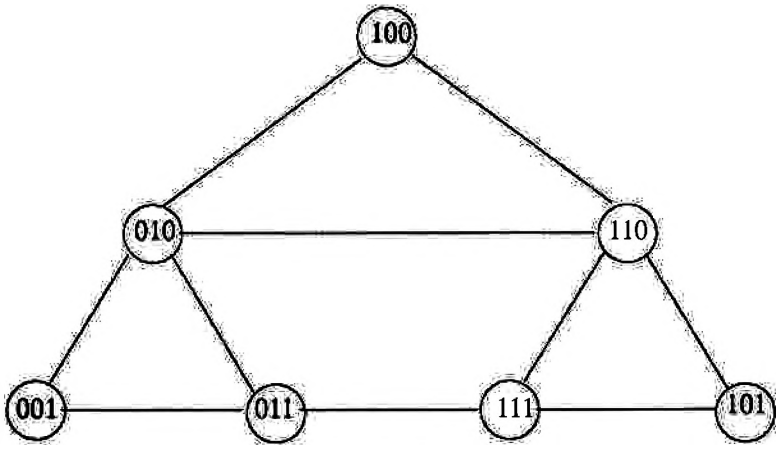


Fig. 22. 2D Pyramid Generated by HBRGC HB

4 Models of Embedding, Partitioning and Mapping

The goal of partioning and mapping of a parallel program onto an architecture is to provide a balanced node utilization by allocating processes to processors maximizing parallelism while, simultaneously reducing communication overhead. These two goals are contradictory. The number of processes assigned to each node is application dependent and is dependent on the ratio between computation and communication time.

Optimal load balancing under perfect information is possible. In this case, you are given a set of processes $\rho_0, \rho_1, ..., \rho_N - 1$ with execution time requirements of $w(\rho_0), w(\rho_1), \cdots, w(\rho_N - 1)$ and a set of communication costs: $\mathbf{C} = C(i,j)$ which is the length of a message sent in communicating from process ρ_i to process ρ_j.

Classically [12], the goal of load balancing, given a process/communication digraph $G(\mathbf{P}, \mathbf{C})$, where $\mathbf{P}$ is the set of processes and $\mathbf{C}$ is the set of directed arcs $C(i,j)$, is to find a partition

$$G = G_0 \cup G_1 \cup ... \cup G_T - 1$$

of G and a mapping of processes to processors $n(\rho)$ subject to the following constraints,

$$W_n = \sum_{p \in G_n} w(\rho) = constant \tag{9}$$

$$C = \frac{1}{2} \sum \sum_{\rho \neq \rho'} C(\rho, \rho') \cdot dist(n(\rho), n(\rho')) is\ minimized \tag{10}$$

The problem with this metric is that, in modern multicomputers, such as the NCUBE/2, Intel Paragon, and CM-5, the time to traverse multiple hops in

the k-ary n-cube is roughly equivalent to the time to perform nearest neighbor communication. Thus, we can simply rewrite Equation 10 as

$$C = \frac{1}{2}\sum\sum_{\rho \neq \rho'} C(\rho, \rho') \text{ is minimized} \tag{11}$$

Intuitively, however, this model is also inadequate for it does not take into account *congestion* from Definition 12. Consider an example of the effects of congestion from a ring embedding of the protein sequence comparison from Section 3.

For simplicity, if we model the communication in a hypercube as circuit switching, then a hardware communication circuit between two communicating nodes must be established before communication begins, and a link of the circuit is released at a time after the last bit of the message is transmitted. We, therefore, define the communication time needed for two communicating nodes in a hypercube as follows,

$$\begin{aligned} t_{comm} &= t_{cong} + t_{hops} \\ &= t_{cong} + [\tau_s + \tau_t C(\rho, \rho')] \end{aligned}$$

where t_{comm} is the time needed to send an C-byte message from one node to another. For the circuit switching model, if a circuit cannot be established because a desired link is being used by other packets, the circuit is said to be blocked. Here we assume that when a circuit is blocked, the partial circuit may be torn down, with establishment to be attempted later. t_{cong} here denotes the waiting time for reestablishment. Note that, if the mapping of the linear array in a hypercube is dilation-1, then it will be congestion-1 also and no edges of a hypercube will be contained in more than one mapping linear array edge. That is, if the mapping is dilation-1, t_{cong}, the communication delay due to congestion, will be zero. t_{hops} is the ideal communication time between two communicating nodes such that the edge congestions of the desired circuit between these two nodes are all one. The value of t_{hops} is determined by the three terms: τ_s, τ_t, and C, where τ_s is the communication latency and τ_t is the time needed to transmit one byte of data. In the parallel protein sequence comparison, each processor in the linear array will send messages to its right neighbor twice, therefore, $T_{comm} = 2 \cdot t_{comm} = 2 \cdot (t_{cong} + t_{hops})$.

Suppose that, during the course of the computation, some processor fails. If in the beginning we select one designated spare node and let the rest of nodes all do the computation. If a node becomes faulty during processing, just replace this faulty node with this designated spare node. For this approach, it is *very* possible that the length (or hops) of the desired path from the left or right neighbor of the faulty node to the designated spare node is equal to the dimension of the embedding hypercube, and, moreover, the desired path has congestion-2. These factors (number of hops and congestion) have to be taken into account for calculating the communication time. From the algorithm of parallel protein sequence comparison, we can derive that t_{cong} is equal to $\tau_s + \tau_t C$. For simplicity,

we also assume that the path from the faulty node's left neighbor to the designated spare node and the path from the designated spare node to the faulty node's right neighbor are edge-disjoint. The total running time for this approach is about,

$$\begin{aligned} t_{cong} &= \tau_s + \tau_t C \\ t_{hops} &= \tau_s + \tau_t C \\ T_{comm} &= 2 \cdot (t_{cong} + t_{hops}) \\ T_\rho &= T_{comm} + w(\rho) \end{aligned}$$

which, by comparison with an embedding with no congestion, essentially, doubles the communication time of the entire problem.

5 A Mathematical Model of Distributed Systems Behavior

A *formal model* (or mathematical model) is a model of the system using well-understood mathematical entities such as sets and functions. Formal methods used in developing computer systems are mathematically based techniques for describing systems. A formal method consists of a formal model and associated mathematical techniques which provides the user with a framework for specifying and analyzing the system.

The problem of specifying an abstract system is that of specifying a particular mathematical object, for which good mathematical techniques may have already been developed over the years. The existence of a formal model of an abstract system implies that a formal statement of the problem is needed that is in terms of the the formal model being used. Separating the problem from its solution is an important contribution of having a theoretical foundation in that it opens the door to alternative solutions [11].

There are numerous examples of the use of mathematical models in the computer science literature. One example from the study of network topology is being able to compute the information carrying capacity of a network. This Graphs can be used as the model of network topology, while the concept of the cut is useful for modeling the carrying capacity of the network. Other examples include queuing models for analyzing the performance of a system, Markov chains for reliability analysis, and axiomatic and denotational specifications for formally describing programming languages.

In general, theoretical foundations can provide (1) criteria for evaluation, (2) means of comparison, (3) theoretical limits and capabilities (4) means of prediction, and (5) underlying rules, principles, and structure. The power of a mathematical model is that it forces one to think clearly about the problem one is trying to solve. The process of stating the question leads one to identify relevant variables, state explicitly any assumption being made, and so forth. These very factors are often instrumental in leading one to a solution. Models ignore irrelevant details. This focuses attention on the essential feature; thus,

a model produces generality, for results that depend on fewer assumptions are more widely applicable.

5.1 The Axiomatic Approach to Program Verification

The axiomatic approach to program verification is based on making assertions about program variables before, during and after program execution. These assertions characterize properties of program variables and relationships between them at various stages of program execution. Program verification requires proofs of theorems of the following type:

$$< P > S < Q >$$

where P and Q are assertions, and S is a statement of the language. The interpretation of the theorem is as follows: if P is true before the execution of S and if the execution of S terminates, then Q is true after the execution of S. P is said to be the *precondition* and Q the *postcondition*[16]. A statement, S, is *partially correct* with respect to the precondition P and a postcondition Q, if, whenever, P is true of S prior to execution, and if S terminates then Q is true of S after the execution of S terminates. A program, S, is *totally correct* if it is partially correct and it can be shown that this program terminates.

CSP programs are composed of a set of communicating sequential processes. In many programs, it is desirable to save part of the communication sequence between processes. This is done with use of "dummy" or *auxiliary* variables that relate program variables of one process to program variables of another. The need for such variables has been independently recognized by many. The first reference that shows the usefulness of auxiliary variables is found in [7].

Overall Proof Approach . As discussed before a CSP program is made up of component sequential processes executing in parallel. In general, to prove properties about the program, first properties of each component process are derived in isolation. These properties are combined to obtain the properties of the whole program.

Example 4. Assume that we want to prove the following:

$$< true > [\rho_1 || \rho_2 || \rho_3] < x = u >$$

where

$$\rho_1 :: \rho_2 ! x$$
$$\rho_2 :: \rho_1 ? y; \rho_3 ! y$$
$$\rho_3 :: \rho_2$$

The following properties can be proven about each of the component processes:

$$< x = z > \rho_1 < x = z >$$
$$< true > \rho_2 < y = z >$$
$$< true > \rho_3 < u = z >$$

We can use the properties that $x = z$ and $u = z$ and transitivity to show that $x = u$.

There are two approaches to proving the correctness of communicating processes. The first approach is to divide the correctness proof into two parts. The first is the sequential proofs of each individual process that makes assumptions about the effects of the communication commands. The second part is to ensure that the assumptions are "legitimate". This will be discussed later. This approach is taken in [2] and [25]. The second approach allows us to prove properties of the individual processes using the axioms and rules of inference applicable to the statements in the individual processes. The axioms and rules of inference are designed in such a way that it is not necessary in a sequential proof of a process to make assumptions about the behavior of other processes. These properties are then used to prove properties of the entire program. This is the approach of [39].

It has been shown [26] that it is irrelevant as to which axiomatic proof systems of program verification is chosen. This was done by showing that the axiomatic systems are equivalent in the sense that they allow us to prove the same properties. No system is more powerful than the other. However, there are very different approaches to thinking about the verification of the program and the applicability in a practical environment. The proof system presented in [25] is presented here for its relative ease of use.

Axioms and Inference Rules Used For Sequential Reasoning . In addition to the axioms and inference rules of predicate logic, there is one axiom or inference rule for each type of statement, as well as some statement-independent inference rules. The following are common to all the axiomatic systems and apply to reasoning about sequential programs. The basis of the axiomatic approach to sequential programming can be found in [16].

The *skip* axiom is simple, since execution of the skip statement has no effect on any program or auxiliary variables.

$$< P > skip < P >$$

The axiom states that anything about the program and logical variables that holds before executing **skip** also holds after it has terminated.

To understand the *assignment* axiom, consider a multiple assignment statement, $\bar{x} := \bar{e}$, where $\bar{x}$ is a list of $x_1, x_2, ..., x_n$ of identifiers and $\bar{e}$ is a list of $e_1, e_2, ..., e_n$ of expressions. If execution of this statement does not terminate, then the axiom is valid for any choice of postcondition P. If execution terminates, then its only effect is to change the value denoted by each target x_i to that of the value denoted by the corresponding expression e_i before execution was begun. Thus, to be able to conclude that P is true when the multiple assignment terminates, execution must begin in a state in which the assertion obtained by replacing each occurrence of x_i in P by e_i holds. This means that if $P^{\bar{x}}_{\bar{e}}$[3] is true before the multiple assignment is executed and execution terminates, then P will be true after the assignment. Thus we have the following:

[3] This stands for predicate P with each x_i replaced with e_i

$$< P^{\bar{x}}_{\bar{e}} > \bar{x} := \bar{e} < P >$$

It may seem strange at first that the precondition should be derived from the postcondition rather than vice versa, but it turns out that this assignment rule, as well as being simple, is very convenient to apply in constructing proofs about programs.

There are also a number of rules of inference, which enable the truth of certain assertions to be deduced from the truth of certain other assertions.

A proof outline for the composition of two statements can be derived from proofs for each of its components.

$$\frac{< P > S_1 < Q >, < Q > S_2 < R >}{< P > S_1; S_2 < R >}$$

When executing $S_1; S_2$, if Q is true when S_1 terminates it will hold when S_2 starts. From the second hypothesis, if Q is true just before S_2 executes and S_2 terminates, then R will hold. Thus if S_1 and S_2 are executed one after the other and P holds before the execution, then R holds after the execution.

Execution of an alternate command ensures that a statement S_i is executed only if its guard b_i is true. Thus, if an assertion P is true before execution of the alternate command, then $P \wedge b_i$ will hold just before S_i is executed. The second part of the hypothesis assumes that none of the guards are true. If the hypothesis is true and if the alternate statement terminates, then this is sufficient to prove that Q will hold should the alternate statement terminate.

$$\frac{\forall i :< P \wedge b_i > c_i; S_i < Q >, < P \wedge \forall i : \neg b_i > \rightarrow < Q >}{< P > \textbf{if} \Box b_i; c_i \rightarrow S_i \textbf{fi} < Q >}$$

The consequence rule allows the precondition of a program or part of a program to be strengthened and the postcondition to be weakened, based on deductions possible in the predicate logic.

$$\frac{P \rightarrow P', < P' > S < Q' >, Q' \rightarrow Q}{< P > S < Q >}$$

The need for auxiliary variables was discussed earlier. Two of the proof systems use auxiliary variables. The auxiliary variables must not affect program control during execution. The following rule allows us to draw conclusions from proof outlines of programs annotated with auxiliary variables.

$$\frac{< P > S' < Q >}{< P > S < Q >}$$

where S is obtained from S' by deleting all references to auxiliary variables and P and Q do not contain any free variables which are auxiliary variables.

The inference rule for the repetition command is based on a loop invariant i.e. an assertion that holds both before and after every iteration of a loop.

$$\frac{\forall i :< P \wedge b_i > c_i; S_i < P >}{< P > *[\Box b_i c_i \rightarrow S_i] < P \wedge \forall i : \neg b_i >}$$

The hypotheses of the rule require that if execution of S_i is begun when the assertion P and b_i is true, and if execution terminates, then P will again be true. Hence, if an assertion P is true just before the execution of a repetition command, then P is true at the beginning and end of each iteration. Thus, P will hold if the repetition terminates. The repetition ends when no boolean guard is true, so $\neg b_1 \wedge \neg b_2 \wedge ... \wedge \neg b_n$ will also hold at that time.

[25] does not have distributed termination which is contrary to Hoare's original version of CSP [17]. Distributed termination provides the means for automatic termination of a loop in one process because another process has terminated. It is assumed that termination of all loops occurs when all boolean guards are false.

Example 5. Let us examine how these rules are applied to the following sample program.

```
var t,i,b[0...n-1]:integer;
t := 0;
i := 0;
do [i ≠ n → t:=t+b[i]; i := i + 1] od
```

This program sums up the elements of an array b. The result is put into the variable t. Now to prove the partial correctness of this program, we will prove that if the program is started in a state where $n \geq 0$ holds and execution terminates, then t will contain the sum of the values in b[0] through b[n-1]. The composition rule implies that in order to prove the above program correct, it is sufficient to prove that

$$< n \geq 0 > t := 0 < t = 0 > \tag{12}$$

$$< t = 0 > i := 0 < t = 0 \wedge i = 0 > \tag{13}$$

$$< t = 0 \wedge i = 0 > \textbf{do}[i \neq n \rightarrow t := t + b[i]; i := i + 1]\textbf{od} < t = \sum_{j=0}^{i-1} b[j] > \tag{14}$$

Outline 12 and 13 are easy to prove using the assignment axiom. Note that using normal predicate logic inference rules, it can be shown $t = 0 \wedge i = 0 \rightarrow t = \sum_{j=0}^{i-1} b[j]$. Remember that since i is equal to 0, that there are no values of j between 0 and $i - 1$. Hence, $t = \sum_{j=0}^{i-1} b[j]$ is vacuously true. Therefore, by applying the Rule of Consequence, we can prove Outline 14 by showing the following:

$$< t = \sum_{j=0}^{i-1} b[j] > \text{do}[i \neq n \rightarrow t := t + b[i]; i := i + 1]\text{od} < t = \sum_{j=0}^{i-1} b[j] > \quad (15)$$

To prove Outline 15, it will be sufficient to prove that

$$< t = \sum_{j=0}^{i-1} b[j] \wedge i \neq n > t := t + b[i]; i := i + 1 < t = \sum_{j=0}^{i-1} b[j] > \quad (16)$$

$$< t = \sum_{j=0}^{i-1} b[j] \wedge i = n > \rightarrow < t = \sum_{j=0}^{n-1} b[j] > \quad (17)$$

In order to prove 16, it is sufficient to prove

$$< t = \sum_{j=0}^{i-1} b[j] \wedge i \neq n > t := t + b[i]; < t = \sum_{j=0}^{i} b[j] > \quad (18)$$

$$< t = \sum_{j=0}^{i} b[j] > i := i + 1 < t = \sum_{j=0}^{i-1} b[j] > \quad (19)$$

Outlines 18, 19 can each be proven by applying the assignment axiom. Outline 17 can be shown by substituting i for n and using the consequence rule.

Axioms and Inference Rules Dealing With Communication . Each of the three proof systems deal with assertions on communications in different manners. Two of the approaches make the explicit use of auxiliary variables to relate the different communication sequences. The third proof system makes assertions on communication sequences.

Communication and Parallel Decomposition rules . The communication axiom is as follows:

$$< P > \beta < Q >$$

where β is a communication command.

Remember that $< P > S < Q >$ means total correctness if S terminates. S terminates in the absence of deadlock. The parallel rule implies that a proof for a parallel program is based on the isolated sequential proofs of the processes it comprises. Take any such program S. A sequential proof for it only proves facts about it running in isolation. With only one process running, communication commands deadlock. Thus, any predicate Q may be assumed to be true upon termination of a communication command because termination never occurs.

The Law of the Excluded Miracle [9] states that the statement false should never be derived. This is the requirement to ensure a sound logic. The communication axiom does violate the Law of the Excluded Miracle. This allows us to deduce that the following is true:

$$< true > A?x < x = 5 \wedge x = 6 >$$

The postcondition, however, is obviously false. Thus, one might come to the conclusion that the proof system is not sound. This is the result of allowing the communication axiom to make assumptions about the behavior of other processes in order to prove properties of an individual process. In order to justify those assumptions a "satisfaction proof" must be done. This ensures that the proof system is sound. Hence, the parallel inference rule is as follows:

$$\frac{(\forall i :< P_i > S_i < Q_i >,)\text{satisfied and interference} - \text{free}}{< (\forall i : P_i) > [||_{i=1:n}, \rho_i :: S_i] < (\forall i : Q_i) >}$$

The parallel rule implies that we can construct the proof of a parallel program from the partial correctness properties of the sequential programs it comprises.

It has been mentioned that a "satisfaction proof" is needed to ensure soundness of the proof system. Let us examine the proof outline of the matching communication pair:

$$\rho_1 : [... < P > \rho_2 ?x < Q >]$$

$$\rho_2 : [... < R > \rho_1 !y < S >]$$

The effect of these two communication commands is to assign y to x. This implies that $Q \wedge S$ is true after communication if and only if

$$(P \wedge R) \rightarrow (Q \wedge S)^x_y$$

A "satisfaction proof" is such that the above is proven for every matching communication pair. This is called the *rule of satisfaction*.

Earlier we discussed the need for auxiliary variables. An auxiliary variable may affect neither the flow of control nor the value of any non-auxiliary variables. Otherwise, this unrestricted use of auxiliary variables would destroy the soundness of the proof system. Hence, auxiliary variables are not necessary to the computation, but they are necessary for verification. The proof system in [25] allows for auxiliary variables to be global i.e. variables that can be shared between distinct processes. Global auxiliary variables (GAVs) are used to record part of the history of the communication sequence. Shared reference to auxiliary variables allow for assertions relating the different communication sequences. This necessitates the need for a *Proof of Non-interference*. This consists of showing that for each assertion T in process ρ_i, it must be shown that T is invariant over any parallel execution. This is the non-interference property of [32].

Asynchronous Message Passing Systems The proof systems that have been discussed up to this point are designed for synchronous programming primitives. Our work uses an extension of work discussed in [37]. The work of [37] describes how to extend the notion of a "satisfaction proof" and "non-interference proof" for asynchronous message-passing primitives. The extension is based on introducing for each pair of processors ρ_i and ρ_j, two auxiliary variables δ_{ij}, γ_{ij}, where δ_{ij} is the set of all messages sent from process i to process j and γ_{ij} is the set of all messages j actually receives from i. This extension involves assuming

that actual sending and receipt of a message implies that δ_{ij} and γ_{ij} are immediately updated. It is also assumed that $\gamma_{ij} \subseteq \delta_{ij}$ is invariantly true throughout program execution.

5.2 Branch and Bound Example

Branch and bound algorithms have been used in the past to search optimal solutions for many well-known problems such as the Traveling Salesman and the N Puzzle Problem. These problems typically correspond to trees or graphs with exponential search space. There exists many search strategies that can find the optimal solution such as breadth-first, depth-first search, and best-first search. We assume that the Branch and Bound uses a best-first search strategy, in that a function is used for estimating the node that is most promising. When a solution is obtained, the value of the function acts as an upper/lower bound to the problem where all other intermediary bounds can be pruned if it exceeds this bound. The algorithm described in this paper is the N Puzzle Problem where $N + 1$ is a perfect square. The description of the problem is described below.

Most people are familiar with the N Puzzle Problem. The game begins with a given board configuration where the tiles are out of order. The objective is to find a solution that takes the minimal moves to go from the initial board configuration to a known final configuration. Figure 23 shows this.

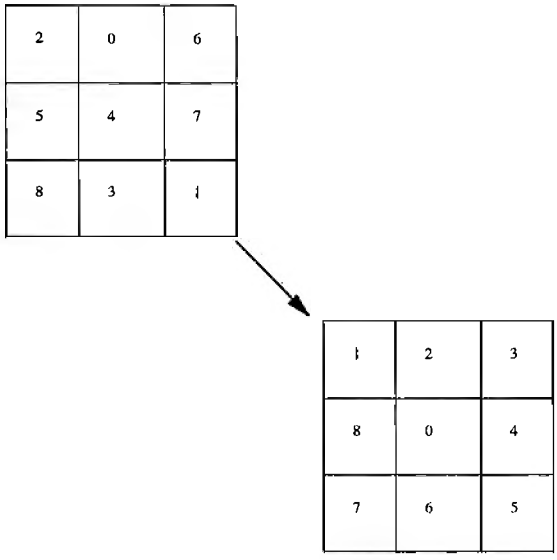


Fig. 23. Objective of the N Puzzle Problem ($N = 8$)

The initial board configuration for the N Puzzle Problem consists of $N + 1$

tile positions with N tiles distinctly numbered ranging from 1 to N and one blank space denoted by a zero.

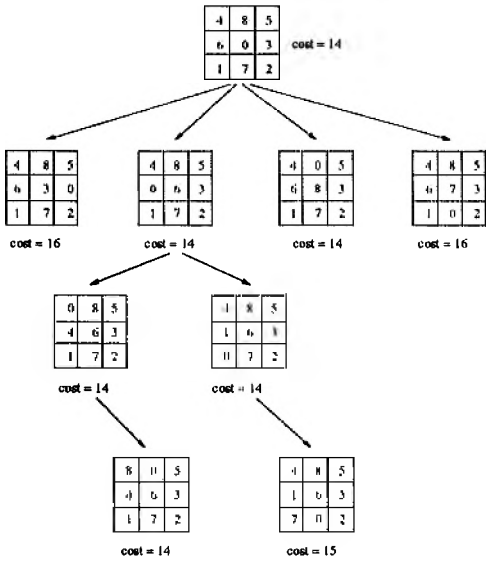


Fig. 24. An abstraction of the N Puzzle Problem (N = 8)

The number of ways of reaching the objective is described best by means of a tree structure. The initial configuration corresponds to the root of a search tree (see Figure 24), where its children are the result of moving an adjacent tile into the blank position. A move which is legal is defined below.

Definition 26. A legal move is described by swapping the blank tile with a tile to its left, right, top or bottom. For each configuration, the following moves are legal if the conditions hold. There exists at most 4 possible moves per configuration. If $b_{u,v}$ is the position of the blank tile then at least two of the following conditions will hold.

$$\begin{aligned} m_0 &: b_{u,v} \leftrightarrow b_{u+1,v} \text{ if} u \neq \sqrt{N+1} \\ m_1 &: b_{u,v} \leftrightarrow b_{u-1,v} \text{ if} u \neq 1 \\ m_2 &: b_{u,v} \leftrightarrow b_{u,v+1} \text{ if} v \neq \sqrt{N+1} \\ m_3 &: b_{u,v} \leftrightarrow b_{u,v-1} \text{ if} v \neq 1 \end{aligned}$$

We will let M be the set $\{m_0, m_1, m_2, m_3\}$.

Each node in the tree is referred to as a state, s_i, of the puzzle where i is a unique integer. We will assume that s_0 denotes the state corresponding to the initial configuration. The state s_i will be represented by the path taken from the initial configuration, s_0 to the node represented by s_i. A path is the sequence of moves from s_0 to the node s_i. This can be formally defined as follows.

Definition 27. Let s_i of the N Puzzle Problem, be represented by the path that exists when a set of moves from the initial configuration, s_0, evolves into the node to be denoted by s_i, where the number of moves taken is k.

$$s_i = (p_{i0}, p_{i1}, p_{i2} \cdots p_{ik-1})$$

where p_{ij} is a move of type m, where $m \in M$.

Definition 28. A node denoted by s_i is a reachable configuration from a node denoted by s_j if s_j is a path prefix of s_i.

If s_i is a reachable configuration from s_j then s_i and s_j are on the same path in the search tree.

Definition 29. Let the search space for the N Puzzle problem be described as follows:

$$S_I = \{s_i | s_i \text{ is a reachable configuration of } s_0\}$$

Definition 30. If $s_i \in S_I$ then s_i is a solution if s_i corresponds to a node which denotes the final configuration.

Definition 31. If a solution exists, then the solution space, A_I, is the set containing all solutions for the N Puzzle Problem.

$$A_I = \{s_i | s_i \in S_I \wedge s_i \text{ is a solution}\}$$

To search an entire tree for the optimal solution is exponential, therefore an optimization function is used to reduce the search space. This optimization function is applied to each path in the tree until the path containing the final configuration of the puzzle is found. The value of this function acts as a bound for that particular path down the tree and aids in selecting the next path to expand.

Definition 32. The optimization function, f, determines the cost for a particular state, s_i, at level k.

$$f(s_i) = md_i + k$$

where md_i is the manhattan distance of the configuration denoted by s_i,

The optimization function for the N Puzzle Problem is defined as the sum of the manhattan distance of each tile plus the height of the tree. This function represents the distance each tile is out of place and the number of moves taken thus far.

Theorem 33 states that if the branch and bound algorithm finds the minimal cost node as defined by the minimum cost function, which satisfies certain properties, then the minimal cost node is also the optimal node.

Theorem 33. *[23] Let s denote any node representing a minimal cost node, according to the cost function f, where f is monotonically nondecreasing. Then s is an optimal node in the search space.*

Theorem 34. *[28] The optimization function, f, is a monotonic nondecreasing function as i increases.*

By Definition 32 and Theorem 34, once a solution is found, the manhattan distance decreases to zero resulting in k for some $f(s_i)$ evaluated at level k. This value corresponds to the number of moves it took to solve the puzzle. The function value then acts as an upper bound to the problem such that all branches which possess a bound higher than the upper bound need not be considered. The nondecreasing nature of the function signifies that any path being considered with a higher bound, will take at least that many moves to solve the puzzle. Since a solution has already been found which can solve the puzzle in fewer moves, that path will not lead to a better solution. However, the remaining paths with lower bounds must continue expanding, generating new states until it reaches a solution or until it exceeds the current upper bound. When all the nodes of the tree have been explored, the solution having the lowest bound is the one with the minimum number of moves. The parallel algorithm described in this paper is based on this concept.

The parallel algorithm involves dividing the work in terms of subtrees and has many processors working on the problem simultaneously. Two approaches are described in this section.

The approach commonly taken in implementing Branch and Bound is the "worker/controller" method. The initial board configuration of the problem, s_0, is given to a designated processor called the "controller" who distributes the work to all idle processors known as "workers". The controller manages the tasks which are to be completed and is responsible for assigning tasks to the workers. A task refers to transforming some s_i to s_j by a legal move of type, m_k, where $m_k \in M$. A worker, on the other hand, is oblivious to the other workers and does only the task which is assigned to him by the controller. When a worker completes his task, he reports back to the controller by sending the controller the new s_j's each with a bound accordingly to the $f(s_j)$. The controller receives these new s_j's from the workers and inserts them into a prioritized queue which is in increasing order by the value of the bound, $f(s_j)$. When the controller recognizes that a particular worker is idle, he assigns the configuration with the lowest bound from his queue to that worker. As soon as a solution is found,

the controller modifies his queue, disregarding all configurations with higher bounds. This process continues until the controller no longer has any tasks left to distribute and all workers are idle. The current solution, $s_{current_i}$ which has the lowest bound then contains the optimal moves for the puzzle.

```
Host:
Begin
    Distribute initial board configuration;
    Terminate workers;
End;
Worker:
Begin
    Wait for a task to work on;
    Loop
        While ( |task_i| > 0)
            Work on expanding lowest costing paths;
            If (first solution found)
            If f(s_task_i) > f(s_current_i)
                Discard task.
            If a solution s_i is found
                Notify other workers of s_i;
            If (|task_i| > 1) and (worker_i = idle)
                Distribute task to idle worker_i;
            If a solution is reported
            If f(s_recv_i) < f(s_current_i)
                s_current_i = s_recv;
        End While;
        Wait for a task to work on;
        If a solution is reported
            If f(s_recv_i) < f(s_current_i)
                s_current_i = s_recv_i;
    End Loop;
End.
```

Fig. 25. Parallel Branch and Bound for N puzzle

The advantage of this scheme is the clear structural organization of the tasks. The controller knows the status of each worker and keeps track of the best bound reported thus far. However, the controller has become the point of centralization. All messages from the workers are directed to the controller creating a bottle neck. In terms of fault tolerance, the idea of having a centralized control is discouraged because should the fault manifest itself within that particular processor appointed to be the controller, recovery would be impossible.

We use as a model algorithm that of [28] The algorithm requires only workers to search the solution space. The initial task, s_0, is assigned a designated worker

to work on. As time passes, more tasks are created. Each worker retains one task for himself and redistributes the rest to other idle workers. When a worker has completed his task, he notifies the other workers that he is available to accept tasks; otherwise, he continues working on the original tasks distributed to him. The only other communication that occurs among the workers is when a solution, s_i has been found. This solution is only one of many in the solution space, A_I, and may not be the best solution, but it allows some pruning to be done such that the number of tasks can be reduced. The worker who discovers the solution broadcasts to the other workers allowing them to update their local bounds. The algorithm terminates when all tasks have either completed or been discarded. Asynchronous communication and dynamic allocation of tasks are used in this method. The algorithm is described Figure 25.

Verification of the N Puzzle Problem The algorithm presented in Figures 5.2 and 5.2, is the algorithm presented in Figure 25 annotated with assertions and assignments to auxiliary variables(denoted in italics).

Not all assertions are listed, since, there is one assertion, labelled I(defined in Definition 35), that is invariant throughout execution, except during communication. This assertion and why it is not invariant throughout execution is described after Figure 5.2.

$< Pre_H >$
Host:
Begin
  $S, A = S_I, A_I$;
  $S', A' = \emptyset, \emptyset$;
  Distribute initial board configuration;
  $< S_0 \cup S_1 \cup ... \cup S_N - 1 = S_0 >$
  Terminate workers;
End;
$< Post_H >$

Fig. 26. Verification proof outline of the N puzzle problem (Host)

The following auxiliary variables are used in the verification proof:

S_I : This is the set of all nodes in the tree that represents the state space
A_I : This is the subset of S_I which contains the nodes that are solution nodes.
S' : This is the set of nodes that have been examined by the algorithm, either directly or by pruning.
A' : This is the set of solution nodes that have been examined by the algorithm, either directly or by pruning.
S : This is the set of nodes that have not been examined by the algorithm.

$< Pre_i >$
Worker:
Begin
Wait for a task to work on;
Loop
While ($|task|i| > 0$)
Work on expanding lowest costing paths;

If first solution found
If $f(s_{task_i}) > f(s_{current_i})$
Discard task.
$T_0 = \{s_i | s_i$ is a reachable configuration of $s_{task_i}\}$
$T_1 = \{s_i | s_i \in T_0 \wedge s_i$ is a solution state$\}$
$S_i, S, S' = S_i - T_0, S - T_0, S' \cup T_1$

If a solution is found
Notify other workers;
Update SB_{ij} for each j;

If ($|task_i| > 1$) and ($worker_i = IDLE$)
$T_0 = \{s_i | s_i$ is a reachable configuration of $s_{task_i}\}$
$S_i = S_i - T_0$
Distribute task to idle $worker_i$;

If a solution is reported
If $f(s_{recv_i}) < f(s_{current_i})$
$s_{current_i} = s_{recv_i}$;
Update R_{ij}, where j is the sending process;

End While;

Wait for a task to work on;
$T_0 = \{s_i | s_i$ is a reachable configuration of $s_{task_i}\}$
$S_i = S_i \cup T_0$

If a solution is reported
If $f(s_{recv_i}) < f(s_{current_i})$
$s_{current_i} = s_{recv_i}$;
Update R_{ij}, where j is the sending process;
End Loop;
$solution_i = s_{current_i}$
End.
$< Post_i >$

Fig. 27. Verification proof outline of the N puzzle problem (Worker)

A : This is the set of solution nodes that have not been examined by the algorithm.
S_i : This is the set of nodes to be examined by process i.
S_i : current task set in process i, empty or not
$task_i$: task set of process i
SB_{ij} : This is the set of solutions sent from i to j.
RB_{ij} : This is the set of solutions received by i from j.
$solution_i$: This is the value of $s_{current_i}$ at the termination of worker i.
$s_{current_i}$: current optimal solution
s_{recv_i} : solution received from another process at process i
s_{task_i} : state representing the task set of process i

The precondition to the host process assumes that there is a solution from the initial configuration. In other words, Pre_H is as follows:

$$< A_I \neq \emptyset >$$

For each terminating component process labelled node, s_node is the lowest cost solution in the search tree. At the termination of the program, we want each processor to have the same lowest bound. Therefore, the postcondition $Post_H$ is represented as follows:

$$< solution_0 = solution_1 = ... = solution_{N-1} \wedge solution_i \text{ is the optimal cost solution} >$$

The precondition, Pre_i to each worker process i is the assumption that the worker processes initially have no tasks to examine and no communication has taken place. This is represented by

$$< S_i = \emptyset \wedge SB_{ij} = \emptyset \wedge RB_{ij} = \emptyset \text{ for } j, 0 \leq j \leq N-1, j \neq i >$$

The postcondition, $Post_i$ of the worker process i, is that the local variable $s_{current_i}$ has the following property:

$$< s_{current_i} \text{ is the optimal cost solution} >$$

Since S and A are the sets of nodes and solutions, respectively, that have yet to be examined; then since, at the beginning of the program none of the nodes or solutions have been examined, S and A are initialized to S_I and A_I respectively. Similarly, since S' and A' are the set of examined nodes and solutions, respectively; then since, at the beginning of the program none of the nodes or solutions have been examined, S' and A' are initialized to $\emptyset$.

The distribution of the initial board configuration corresponds to assigning to each process i, a subset of the nodes in the tree that represents the state space. No two processes should examine the same nodes. This corresponds to partitioning the auxiliary variable S_I into the disjoint sets $S_0, S_1, ..., S_{N-1}$, which also satisfy the following immediately after initial board distribution:

$$< S_0 \cup S_1 \cup ... \cup S_N - 1 = S_I >$$

In other words, all the search space nodes are distributed among all the worker process nodes. Since, the details of the board distribution are not included in the pseudocode, nor will the verification details.

Each worker process i must only examine those nodes that are part of the state space. Process i is presumed to examine nodes it either (1) Generated through expansion of other nodes or (2) received from other processes. The first case implies that each new generated task must be part of the search space. For the sake of brevity, the details are not shown here, but it involves showing that each newly generated task is a reachable configuration from s_0 and hence, by Definition 29 considered to be an element of the search space represented by S_I. We can conclude that the following is always true:

$$< s_{task_i} \in S_I >$$

Case 2 requires showing that the assertion is true after communication takes place i.e. the satisfaction proof. The details are not described, but it is intuitively easy to see by remembering that i will receive nodes to be examined from other processes, j. We know that $\delta_{ji} \subseteq S_I$ is true, since δ_{ji} is immediately updated as the node is migrated from process j to process i and we know that all nodes migrated are members of S_I (Note that the definition of solutions implies that all solutions are members of S_I). Earlier it was noted that $\gamma_{ji} \subseteq \delta_{ji}$ is invariantly true. We can immediately conclude that

$$< \gamma_{ji} \subseteq S_I >$$

Since, process i must examine only those nodes that are part of the search space then the following must be invariantly true:

$$< S_i \subseteq S_I > \tag{20}$$

There are three instances when S_i is updated:

1. When a part of the search tree is pruned off.
2. When process i receives a node for expansion.
3. When process i migrates a node to another process j for expansion by process j.

For case (1), S_i is changed by first determining the set of nodes associated with the subtree to be pruned off. If the root node of the subtree to be pruned off is s_i, then the set of nodes associated with the subtree to be pruned off is $T_0 = \{s_j | s_j \text{ is a reachable configuration of} s_i\}$. This set of elements is deleted from S_i, i.e. the following operation is done: $S_i = S_i - T_0$. It is, therefore, easy to see that $S_i \subseteq S_I$ is still true.

Showing that cases (2) and (3) maintain the truth of $S_i \subseteq S_I$ is part of the satisfaction proof. Intuitively, this requires showing that the communication maintains the truth of $S_i \subseteq S_I$. For case (2), process i receives a new task from process j. It was earlier shown that that $\gamma_{ij} \subseteq S_I$. For case (2), S_i is updated by first determining the set of nodes associated with the subtree in

which the received node is the root. This is done by determining all the reachable configurations from the received nodes. This set of nodes is added to S_i. Since the received node is a member of S_I then all the reachable nodes from the received node are also members of S_I. Hence, the truth of $S_i \subseteq S_I$ is maintained. A similar argument can be made for case (3).

It is assumed that each node of the state space to be examined is assigned to a process. This is represented as follows:

$$< S_0 \cup ... \cup S_{N-1} \cup \{s | s \in \delta_{ij} - \gamma_{ij} \wedge s \text{ is a task, where} 0 \leq i, j \leq N-1, i \neq j\} = S > \quad (21)$$

Task migration does not change the set of nodes to be examined as a whole. Instead, a task migration only changes the set of nodes to be examined by the migrating and receiving worker processes. These changes were discussed in the previous discussion. Because of the asynchronous nature of the algorithm, it is possible for process i to send a task to process j, but j is not ready to immediately receive the task. Therefore, it is necessary to include the set

$$\{s | s \in \delta_{ij} - \gamma_{ij} \wedge s \text{ is a task, where } 0 \leq i, j \leq N-1, i \neq j\}$$

It is necessary to ensure that the algorithm only examines those nodes that are in the state space. It is also necessary to ensure that all tasks and solutions are examined before they are discarded. This can be ensured by having the following invariantly true:

$$< S' \cup S = S_I \wedge A' \cup A = A_I > \quad (22)$$

The truth of this can be seen by observing that (1) The assertion stated in 22 is true at the beginning of program execution of the worker processes and (2) S, S', A and A' are updated when a subtree is pruned from the search space. The updating is done by determining all the set of nodes associated with the pruned subtree by finding all the reachable nodes from the root node of the subtree to be pruned off and determining all the solution nodes in the pruned subtree. The set of all nodes associated with the pruned subtree is deleted from S and added to S'. The solution nodes are deleted from A and added to A'. These updates are done simultaneously. Since, the assertion is initially true, then from the updates, it is obvious that the assertion stated in 22 is always true.

In the program, in each process i, $s_{current_i}$ is the solution that is known by process i to have the lowest bound. $s_{current_i}$ is changed as more information about other solutions becomes known from other processes. Therefore, the following assertion is invariantly true:

$$< s_{current_i} = min(a | a \in R_i, where R_i = \cup RB_{ij}) > \quad (23)$$

We would like to also ensure that the set of solutions received by each process by the termination of the program is equivalent. The following assertions aid in this. It must be invariantly true except when a solution is being broadcast (only because the auxiliary variable update is done after the broadcast).

$$< SB_{ij} = R_{ji} \cup \{x | x \in \delta_{ij} - \gamma_{ij} \wedge x \text{ is a solution}\} > \quad (24)$$

$$< SB_{i0} = ... = SB_{iN-1} > \tag{25}$$

The assertion stated in 24 states that the solutions that are sent from process i to process j are the same received by process j from process i except for those solutions that are in transit. The assertion stated in 25 is true because process i sends a solution to all processes.

Since, A' is the set of solution nodes known not to be a lower bound, then for each solution node in A' there is at least one solution node in the set of broadcast solutions that is of lesser cost. Mathematically, this can be described as follows:

$$< \forall x \in A', \text{ there is an } i \text{ and } y \text{ such that } y \in R_i \text{ and} f(y) \leq f(x) > \tag{26}$$

Definition 35. Let the assertion I denote the conjunction of the logical expressions expressed in 20-26.

The verification proof shows that I is invariantly true throughout user execution except for the part expressed in (24), (25). However, this is immediately rectified after the communication through assignments to S_{ij}.

The inner loop invariant for process i is the following:

$$< ((task_i > 0 \wedge S_i \neq \emptyset) \vee (task_i = 0 \wedge S_i = \emptyset)) > \tag{27}$$

The outer loop invariant for worker process i is the following:

$$< (worker_i = busy \wedge S_i \neq \emptyset) \vee (worker_i = idle \wedge S_i = \emptyset \wedge \delta_{ij} - \gamma_{ij} = \emptyset) > \tag{28}$$

It can be shown that the termination of the Host process implies that each worker processor knows the optimal cost node. First, it is shown that the termination of the program in Figure 25 implies that each processor received the same set of broadcast solutions.

Lemma 36. *At the termination of the program in Figure 25 the following assertion is true:*

$$< R_0 = \cdots = R_{N-1} >$$

Proof. The outer loop invariant and termination of all worker processes implies that we have $\delta_{ij} - \gamma_{ij} = \emptyset$. Hence, for a process i and any worker process j, we have that $SB_{ij} = RB_{ij}$. From (26), we can conclude that at the termination of the program that $RB_0 i = ... = RB_N - 1i$. This is true for any worker process i. Since, for any worker process j, $R_j = \cup RB_{ji}$, then we can conclude that

$$R_0 = .. = R_{N-1} \square$$

Now, Lemma 36 and Theorem 33 can be used to show that the termination of the program in Figure 25 implies that each worker processor has found the optimal solution.

Theorem 37. The termination of the program in Figure25 implies the postcondition, $Post_H$

Proof. At the termination of the outer loop, we have that all processes are idle and hence, for all i, where $0 <= i <= N-1$, we have that $S_i = \emptyset$. Since, the verification proof shows that $S_0 \cup S_1 ... \cup S_N - 1 = S$, then we can conclude that $S = \emptyset$. Since, $A \subseteq S$ and $A \cup A' = A_I$, then we can conclude that $A' = A_I$. From (23), (26) and Lemma 36 we can easily conclude that $s_{current_i}$ is the same in each process and is the minimal cost as defined by the optimization function defined in Definition 32. From Lemma 36 and Theorem 33, we know that this minimal cost is the optimal cost. □

5.3 Temporal Reasoning

A system is responsive [29] if it responds to internal programs or external inputs in a timely, dependable and predictable manner. It is a necessity for a responsive system to manage initiation and termination of activities to meet the specified timing constraints. Also a responsive system is dependable and predictable, i.e. the system behaves as anticipated, and the occurrence of undesired actions (e.g. faults) does not necessarily lead to a failure.

The incorporation of real-time and fault tolerance into distributed parallel environments is a challenging task, while the specifications of the distributed system must be met within the deadlines in spite of the presence of failures. However, this integrated system or responsive computer system [29] can greatly benefit from the application of formal methods. Without formal techniques, life-critical distributed computer control programs cannot be relied on to produce correct results, in time, and, in the presence of failures.

The liveness assertion $(p \rightarrow \mathrm{EF}q)$ ensures what values the program variables must possess *eventually* at a state with assertion q, starting from a state with assertion p. These assertions are derived from the temporal proof system of Interleaving Set Temporal Logic [33], and are used to reason about *progress* property from one communication point to another.

Since the evaluation of any assertion is, essentially, a check of a safety property, it might be questionable whether, given $(p \rightarrow \mathrm{EF}q)$, it is really necessary to evaluate the assertions p and q in time order. To see why, consider the following example. In a poker game, it is possible, during an evening, to be \$10 in debt and \$10 ahead many times. However, from the point of view of a player it is much better to be ahead \$10 at the end of the night, rather than \$10 in debt! In our logic, we would like to express $p(=\$10\,\text{behind}) \rightarrow \mathrm{EF}q(=\$10\,\text{ahead})$ and ensure, during this poker game, that this temporal assertion is met. If we don't evaluate this assertion in time order, then p and q may be met, but p may occur last which means that we leave the game \$10 poorer, a sorry proposition.

Interleaving Set Temporal Logic Among many mathematical models, we choose Interleaving Set Temporal Logic(ISTL*) [22, 33], since it is capable of representing intermediate behavior of distributed programs.

Traditionally, concurrent and distributed programs are verified using variants of temporal logics with interleaving semantics [6, 5, 30, 31, 15]. However, verification using sets of state sequences which represent the executions of a program is tedious and unnatural since all the possible interleavings of a program must be checked. Thus, many attempts have been made to design a temporal logic with appropriate formalism for distributed programs. That is the partial order approach[21, 34, 35, 33], where the order is defined by the local events (events are executions of atomic operations) within a process and the events of sending and later receiving a message. All other events that are not related are arbitrary.

Interleaving Set Temporal Logic(ISTL*) is based on a partial order approach; thus, the assertions derived are capable of describing the distributed nature of a program. These assertions can also serve as expected behavior to monitor the run time behavior of a program.

In the application to programs, an ISTL* [KaPe88, PePn90] structure corresponds to a computation (run) of a program and each (branching) structure denotes the global states of a single partial order as well as causal relations among these states. An ISTL* structure M is a quadruple $< \Sigma, \theta, E, L >$ where

- Σ is a set of states,
- θ denotes the initial state of Σ,
- E is a set of sequences of states starting at θ ,
- L is a function which assigns to each state $s \in \Sigma$ an interpretation.

A liveness assertion of the form $(p \rightarrow \mathrm{EF}q)$ states that every computation contains some state sequence (path) eventually satisfying the assertion q when starting from a state satisfying the assertion p.

Proof rules of ISTL This section presents the proof rules of ISTL*; for details the reader may refer to [22, 33].

SS-TRANS: This rule handles the case where progress is made by one single operation and derives the temporal formula of the form $\mathrm{EX}q$ from premises which are all state formulas.

$$\frac{\begin{array}{l}(1)\ p \rightarrow \phi \\ (2)\ \phi \rightarrow enabled(\tau) \\ (3)\ \{\phi\}T - \tau\{\phi \vee q\} \\ (4)\ \{\phi\}\tau\{q\}\end{array}}{p \rightarrow \mathrm{EX}q}$$

To justify this rule, suppose that the premises are all valid, consider a computation with a p-state at Σ_i. By premise (1), p implies ϕ and from premises (2) and (3), ϕ holds at Σ_i and all subsequent states until a q-state is reached. Hence a q-state is reached or the transition τ is taken, which results in a q-state by premise (4). The desired conclusion follows.

TRANS: This transitivity rule combines a finite number of successive constraints into a more complicated property. The state formula r is called the link of the rule.

$$\frac{\begin{array}{l}p \rightarrow \mathrm{EF}r \\ r \rightarrow \mathrm{EF}q\end{array}}{p \rightarrow \mathrm{EF}q}$$

CONF: This confluent rule allows us to prove eventual properties that result from combining a number of parallel eventual assertions.

(1) $p \rightarrow \mathrm{EF}(r \vee s)$
(2) $r \rightarrow \mathrm{EF}q$
(3) $s \rightarrow \mathrm{EF}q$

$p \rightarrow \mathrm{EF}q$

To justify this rule, assume that the premises are all valid, and consider an arbitrary computation containing a p-state. By premises (1), a p-state is followed by an r-state or an s-state, both of which are followed by a q-state. The conclusion follows.

Rule WIND: This rule applies induction over well-founded sets, which is commonly used in liveness proofs. Assume that the variable α ranges over the natural numbers **N** and $n \in \mathbf{N}$.

(1) $p \rightarrow \phi(n)$
(2) $(\forall i)(\phi(\alpha) \wedge \alpha = i \wedge i > 0 \rightarrow$
$\mathrm{EF}(\phi(\alpha) \wedge (\exists j)(\alpha = j \wedge i > j)))$
(3) $\phi(0) \rightarrow q$

$p \rightarrow \mathrm{EF}q$

$\phi(\alpha) \wedge \alpha = i$ denotes that the state formula which results from the invariant $\phi(\alpha)$ by replacing all occurrences of the variable α with the value i. To justify this rule, suppose that the premises are all valid, and consider a computation containing a p-state Σ_{k_n}. By premise (1), $\phi(n)$ holds at σ_{k_n} From premise (2), it follows that there is a sequence of positions $k_n \geq k_{n-1} \geq \ldots \geq k_0$, such that each σ_{k_i} $(0 \leq i \leq n)$ is a $\phi(i)$-state and the index α is decreasing in that sequence. In other words, there exists a sequence of assertions, $\sigma_{k_n}, \sigma_{k_n-1}, \ldots, \sigma_{k_0}$ where the index is decreasing.

5.4 Branch and Bound Termination

Since, in this example, all the processes are doing the same computations, we only show the verification involving P_i and P_j. The rest of the proofs are symmetric and can be derived similarly. Note that for the synchronized communications between P_i and P_j where $(i < j)$ P_i will send its message to P_j first and wait for the message from P_j. On the contrary, P_j will not send its message to P_i until it receives the message from P_i. For clarity, superscript i is used to identify a variable being a local variable of process i.

Theorem 38. *$\phi_1 \rightarrow \mathrm{EF}\phi_2$, where $\phi_1 = at(l_0^i) \wedge at(l_0^j)$, $\phi_2 = (at(l_1^i) \wedge at(l_1^j) \wedge (task_j = task_0))$, $(task_j = task_0)$ denotes that P_j receives its task set from P_i. This theorem shows that P_i retains $task_i$ and distributes $task_0$ to P_j.*

Proof. Let θ be the initial condition for the Branch and Bound algorithm. Thus, θ implies ϕ_1. First, to prove $\theta \rightarrow \mathrm{EF}(at(l_0^i) \wedge at(l_0^j))$, by Rule CONF it suffices to show the premises

(1) $\theta \rightarrow \mathrm{EF}(at(l_0^i) \vee at(l_0^j))$
(2) $at(l_0^i) \rightarrow \mathrm{EF}(at(l_0^i) \wedge at(l_0^j))$
(3) $at(l_0^j) \rightarrow \mathrm{EF}(at(l_0^i) \wedge at(l_0^j))$

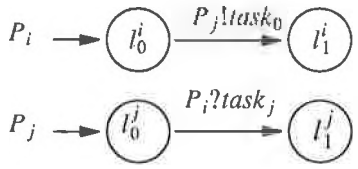


Fig. 28. $\phi_2 \rightarrow \mathrm{EF}\phi_3$

The above three premises are valid since initially the control of P_i and P_j resides in l_0^i and l_0^j, respectively. Then the matching synchronous communication transitions establish the desired conclusion.

Theorem 39. $\phi_2 \rightarrow \mathrm{EF}\phi_3$, *where* $\phi_2 = at(l_1^i) \wedge at(l_1^j)$, $\phi_3 = (S - at(l_4^i) \wedge at(l_4^j) \wedge (s_{recv_j} = s_{task_i}) \wedge (s_i = s_j) \wedge (\alpha' < \alpha))$. $(s_i = s_j)$ *denotes that process* P_j *is informed of the current task set in process* P_i, $(s_{recv_j} = s_{task_i})$ *asserts that* P_j *receives the current best bound of* P_i, *and* $(\alpha' < \alpha)$ *represents the total number of unexplored states is decreasing. This theorem shows that each process works independently on its lowest cost path(expansion), and broadcasts when a solution is found.(See Figure 29)*

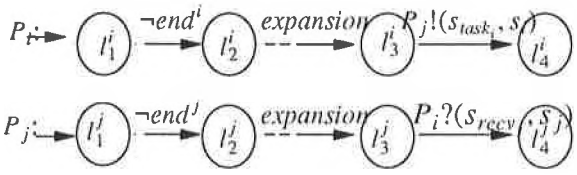


Fig. 29. Each Process Sends a Solution

Proof. Let the assertion *start* hold initially before the computation. Thus, *start* implies ϕ_2. To establish *start* $\rightarrow \mathrm{EF}(at(l_3^i) \wedge at(l_3^j))$ by Rule CONF it suffices to show the following premises

(1) *start* $\rightarrow \mathrm{EF}(at(l_1^i) \vee at(l_1^j))$
(2) $at(l_1^i) \rightarrow \mathrm{EF}(at(l_3^i) \wedge at(l_3^j))$
(3) $at(l_1^j) \rightarrow \mathrm{EF}(at(l_3^i) \wedge at(l_3^j))$

Premise (1) is obviously valid and premise (2) says that P_i will progress to $at(l_3^i)$. This premise can be derived by the transitive rule (TRANS). It suffices to show the premises: (a)*start* $\rightarrow \mathrm{EX}at(l_2^i)$ and (b)$at(l_2^i) \rightarrow \mathrm{EF}at(l_3^i)$. Premise (a) is derived according to the following premises via single step progress rule (SS-TRANS).

$start \rightarrow at(l_1^i)$
$at(l_1^i) \rightarrow enabled(\tau)$
$\{at(l_0^i)\}T - \tau\{at(l_1^i) \vee at(l_2^i)\}$
$\{at(l_1^i)\}\tau\{at(l_2^i)\}$

All of them are trivially valid with respect to one single transition $\tau = l_1^i \rightarrow l_2^i$. Notice that the transitions between l_2^i and l_3^i are not detailed, thus, we omit the proof of $(\alpha' < \alpha)$ from these sequential and non-communication transitions. However, since the set of all nodes to be examined is N^N and each expansion either examines or prunes some nodes, the set of nodes that have not been examined is decreasing.

Premise (3) is similar to premise (2). Thus, we establish $start \rightarrow \mathrm{EF}(at(l_3^i) \wedge at(l_3^j) \wedge (\alpha' < \alpha))$. By synchronous communication transitions we derive $at(l_3^i) \wedge at(l_3^j) \rightarrow \mathrm{EF}(at(l_4^i) \wedge at(l_4^j) \wedge (s_{recv_j} = s_{task_i}) \wedge (s_i = s_j) \wedge (\alpha' < \alpha))$. The conclusion follows.

Theorem 40. *$\phi_3 \rightarrow EF\phi_4$, where $\phi_3 = at(l_4^i) \wedge at(l_4^j)$, $\phi_4 = (at(l_5^i) \wedge at(l_6^j) \wedge (s_{recv_i} = s_{task_j}) \wedge (s_j = s_i))$. $(s_{recv_i} = s_{task_j})$ asserts that P_i receives the current best bound of P_j, and $(s_j = s_i)$ denotes that P_i is informed of the current task set in P_j. This example shows that P_i receives the current best bound and the current task set of P_j. (See Figure 30)*

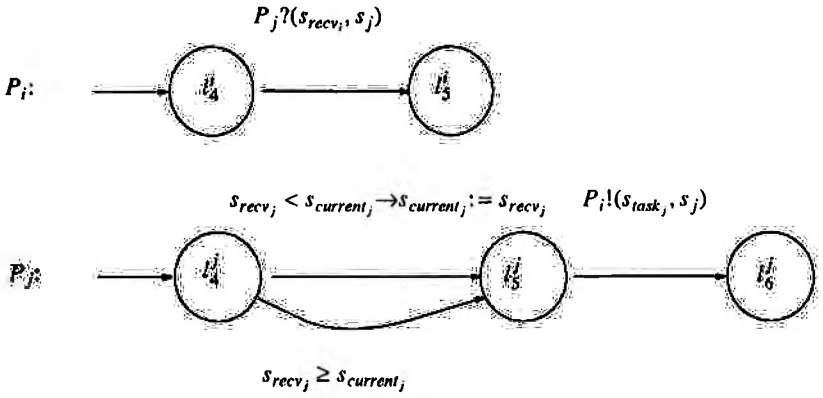


Fig. 30. P_i Receives the Current Best Bound and Task Set of P_j

Proof. Let the assertion *start* hold initially before the execution. Thus, *start* implies Φ_3. To prove $start \rightarrow \mathrm{EF}(at(l_4^i) \wedge at(l_5^j))$, we first apply confluent rule (CONF). It suffices to show the premises

(1) $start \rightarrow \mathrm{EF}(at(l_4^i) \vee at(l_4^j))$
(2) $at(l_4^i) \rightarrow \mathrm{EF}(at(l_4^i) \wedge at(l_5^j))$
(3) $at(l_4^j) \rightarrow \mathrm{EF}(at(l_4^i) \wedge at(l_5^j))$

Premise (1) and (2) are valid since initially the control of P_i and P_j are at l_4^i and l_4^j respectively. Premise (3) requires a case analysis on $(s_{recv_j} < s_{current_j})$ that determines which path is taken. The assertion $at(l_4^j) \rightarrow \mathrm{EF}(at(l_5^j))$ is implied by the following two formulas:

$at(l_4^j) \wedge (s_{recv_j} < s_{current_j}) \rightarrow \mathrm{EF}(at(l_5^j) \wedge (s_{current_j} = s_{recv_j}))$

$at(l_4^j) \wedge (s_{recv_j} \geq s_{current_j}) \rightarrow \mathrm{EF}(at(l_5^j))$

Each of them can be derived via single step progress rule (SS-TRANS) with respect to the transition $\tau = l_4^j \rightarrow l_5^j$. Thus, we establish $start \rightarrow \mathrm{EF}(at(l_4^i) \wedge at(l_5^j))$. Then synchronous communication transition establishes $at(l_4^j) \wedge at(l_5^j) \rightarrow \mathrm{EF}(at(l_5^i) \wedge at(l_6^j) \wedge (s_{recv_i} = s_{task_j}) \wedge (s_j = s_i))$, and, thus, derives the conclusion.

Theorem 41. $\phi_4 \rightarrow \mathrm{EF}\phi_5$, *where* $\phi_4 = at(l_5^i) \wedge at(l_6^j)$, $\phi_5 = ((at(l_1^i) \wedge at(l_1^j)) \vee (at(l_7^i) \wedge at(l_7^j)))$. *This theorem shows task migration. (See Figure 31)*

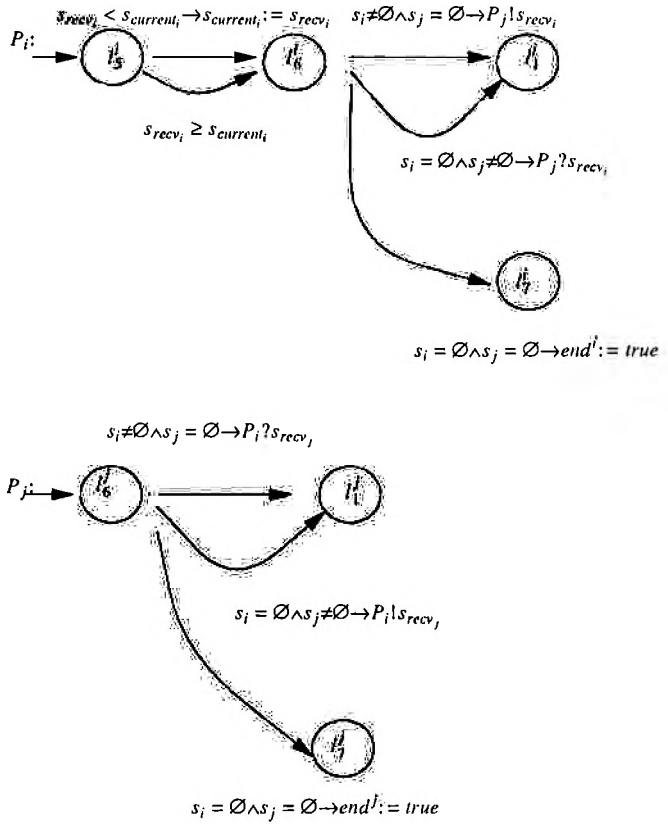


Fig. 31. Task Migration.

Proof. Let the assertion *start* hold initially before the computation. Thus, *start* implies ϕ_4. First to prove $start \rightarrow \mathrm{EF}at(l_6^i) \wedge at(l_6^j)$ by rule CONF, it suffices to show the following premises

(1) $start \rightarrow \mathrm{EF}(at(l_5^i) \vee at(l_6^j))$
(2) $at(l_5^i) \rightarrow \mathrm{EF}(at(l_6^i) \wedge at(l_6^j))$
(3) $at(l_6^j) \rightarrow \mathrm{EF}(at(l_6^i) \wedge at(l_6^j))$

Premise (1) is obviously valid and premise (3) is valid due to the synchronization communication requirement. Premise (2) is implied by the following formulas

$at(l_5^i) \wedge (s_{recv_i} < s_{current_i}) \rightarrow \mathrm{EF}(at(l_6^i) \wedge (s_{current_i} = s_{recv_i}))$
$at(l_5^i) \wedge (s_{recv_i} \geq s_{current_i}) \rightarrow \mathrm{EF}(at(l_6^i))$

Each of them can be derived by single step transitive reasoning (SS-TRANS). To conclude Φ_5 requires a case analysis to determine which path is taken.

There are three cases to consider:

(1) $(at(l_6^i) \wedge at(l_6^j) \wedge s_i \neq \emptyset \wedge s_j = \emptyset \rightarrow \mathrm{EF}(at(l_1^i) \wedge at(l_1^j))$
(2) $(at(l_6^i) \wedge at(l_6^j) \wedge s_i = \emptyset \wedge s_j \neq \emptyset \rightarrow \mathrm{EF}(at(l_1^i) \wedge at(l_1^j))$
(3) $(at(l_6^i) \wedge at(l_6^j) \wedge s_i = \emptyset \wedge s_j = \emptyset \rightarrow \mathrm{EF}(at(l_7^i) \wedge at(l_7^j))$

which may be concluded by SS-TRANS for synchronized communication transitions.

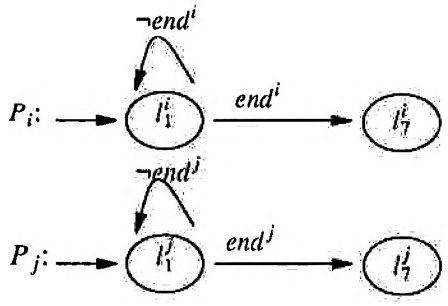


Fig. 32. This diagram shows that either the processes continuously execute the loops or the exit transitions $l_1^i \rightarrow l_7^i$ and $l_1^j \rightarrow l_7^j$ are taken. Define $enter(l) = at(l) \wedge completed(\tau)$.

Theorem 42. *Show termination of the Branch and Bound program, i.e.,* $\phi_2 \rightarrow \mathrm{EF}\phi_6 : (at(l_1^i) \wedge at(l_1^j)) \rightarrow \mathrm{EF}(at(l_7^i) \wedge at(l_7^j))$

Proof. Let *start* hold initially before the computation. Applying the WIND rule, it suffices to show

(1) $start \rightarrow enter(l_1^i) \wedge enter(l_1^j) \wedge (\exists n, \alpha = n)$
(2) $(\forall i)(enter(l_1^i) \wedge enter(l_1^j) \wedge (\alpha = i \wedge \alpha > 0) \rightarrow \mathrm{EF}(enter(l_1^i) \wedge enter(l_1^j) \wedge (\exists j)(\alpha = j \wedge j < i)))$
(3) $enter(l_1^i) \wedge enter(l_1^j) \wedge (\alpha = 0) \rightarrow \mathrm{EF}(enter(l_7^i) \wedge enter(l_7^j))$

Premise (1) is trivially valid. Premise (3) can be concluded by SS-TRANS rule with respect to the transitions $\tau = l_1^i \rightarrow l_7^i$ and $\tau = l_1^j \rightarrow l_7^j$, which says that the control resides at the beginning of the loop and the loop condition is false, thus, the exit transitions are taken. Premise (2) which asserts that the total number of unexplored states is decreasing at each iteration can be established by applying transitivity rule over a finite number of successive properties of $\phi_2 \rightarrow \mathrm{EX}\phi_3$, $\phi_3 \rightarrow \mathrm{EX}\phi_4$, and $\phi_4 \rightarrow \mathrm{EX}\phi_5$.

6 Summary

This paper has covered a broad expanse of topics in an effort to provide both an informal basis for constructing parallel applications and a formal basis for reasoning about these parallel applications and how they are mapped onto a popular existing architecture.

References

1. G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings*, 30:483–485, 1967.
2. R. Apt and W. Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385, 1981.
3. J. Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1979.
4. W. Briggs. *Multigrid Tutorial.* Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1987.
5. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specification. *ACM Transactions on Programming Language and Systems*, 8(2):244–263, April 1986.
6. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications:a practical approach. *10th annual ACM Symposium on Principles of Programming Language*, pages 117–126, 1983.
7. M. Clint. Program proving: coroutines,. *Acta Informatica*, 2:50–63, 1973.
8. G. Cybenko, D. W. Krumme, and K. N. Venkataraman. Fixed hypercube embedding. *Information Processing Letters*, 25:35–39, 1987.
9. E. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Inc., 1976.
10. E. Edmiston and R. A. Wagner. Parallelization of the dynamic programming algorithm for comparison of sequences. In *Proceedings of Int'l Conf. on Parallel Processing*, pages 78–80, 1987.
11. M. Fischer. A theoretician's view of fault tolerant distributed computing. *Fault-Tolerant Distributed Computing, Lecture Notes in Computer Science 448*, pages 1–9, 1990.
12. G. C. Fox and W. Furmaski. Load balancing loosely synchronous problems with a neural network. Technical report, California Institute of Technology, Pasedena, CA, February 1988.

13. W. M. Gentleman. Some complexity results for matrix computations on parallel computers. *Journal of the ACM*, 25(1):112–115, January 1978.
14. J. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.
15. T.A. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for real-time systems. *18th Annual ACM Symposium on Principles of Programming Language*, pages 353–366, 1990.
16. C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.
17. C. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.
18. J. Hong. *Computation, Computability, Similarity and Duality*. Pittman, London, 1986.
19. K. Hwang and Briggs F. *Computer Architecture and Parallel Processing*. McGraw-Hill, New York, 1984.
20. IBM. *IBM Scalable PowerParallel System 9076-SP1*, 1993.
21. S. Katz and D. Peled. Interleaving set temporal logic. *3rd Annual ACM Symposium on Principles of Distributed Computing*, pages 178–190, 1987.
22. S. Katz and D. Peled. An efficient verification method for parallel and distributed programs. *Lecture Notes in Computer Science 354*, pages 489–507, 1988.
23. W. Kohler and K. Steiglitz. Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems. *Journal of the ACM*, 21(1):140–156, 1974.
24. E. Lander and J. P. Mesirov. Protein sequence comparison on a data parallel computer. In *Proceeding of the International Conf. on Parallel Processing*, pages 257–263, 1988.
25. G.M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15:281–302, 1981.
26. H. Lutfiyya and B. McMillin. Comparison of three axiomatic proof systems. *UMR Department of Computer Science Technical Report CSC91-13*, 1991.
27. H. Lutfiyya, B. McMillin, P. Poshyanonda, and C. Dagli. Composite stock cutting through simulated annealing. *Journal of Mathematical and Computer Modeling*, 16(1):57–74, 1992.
28. H. Lutfiyya, A. Sun, and B. McMillin. A fault tolerant branch and bound algorithm derived from program verification. In *IEEE Computers Software and Applications Conference(COMPSAC)*, pages 182–187, 1992.
29. M. Malek. Responsive systems: A challenge for the nineties. In *Proc. EUROMICRO'90, 16th Symp. in Microprocessing and Microprogramming*, pages 622–628, Amersterdam, The Netherlands, 1990. North Holland.
30. Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal proof principles. *Lecture Notes in Computer Science 131*, pages 215–273, 1982.
31. Z. Manna and A. Pnueli. Verification of concurrent programs: A temporal proof system. *Lecture Notes in Computer Science 354*, 1989.
32. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.
33. D. Peled and A. Pnueli. Proving partial order liveness properties. *17th Colloquium on Automata, Language and Programming*, pages 553–571, 1990.
34. S.S. Pinter and P. Wolper. A temporal logic for reasoning about partially ordered computations. *3th Annual ACM Symposium on Principles of Distributed Computing*, pages 28–37, 1984.

35. W. Reisig. Toward a temporal logic for casuality and choice in distributed systems,. In J.W. de Bakker, W.P. Roever, and G. Rozenburg, editors, *Models of Concurrency:Linear, Branching and Partial Orders, LNCS354*, pages 603–627. Springer-Verlag, 1988.
36. D. Riggins, B. McMillin, M. Underwood, L. Reeves, and E. Lu. Modeling of supersonic combustor flows using parallel computing. *Computer Systems in Engineering*, 3:217–219, 1992.
37. R. Schlichting and F. Schneider. Using message passing for distributed programming: Proof rules and disciplines. *ACM Transactions on Programming Languages and Systems*, 6(3):402–431, July 1984.
38. H.J. Siegel and et. al. Pasm: A partionable smid/mimd system for image processing and pattern recognition. *IEEE Transactions on Computers*, C-30:934–947, December 1981.
39. N. Soundararahan. Axiomatic semantics of communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 6(6):647–662, 1984.
40. Q. F. Stout. Hypercubes and pyramids. In V. Cantoni and S. Levialdi, editors, *Pyramidal Systems for Computer Vision*. Springer-Verlag, New York, 1986.

This article was processed using the LaTeX macro package with LLNCS style