Parallel Algorithm Fundamentals and Analysis CSC 93-17

Bruce McMillin *[1], Hanan Lutfiyya **[2], Grace Tsai[1], 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

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. The realities of algorithm design are presented through partitioning and mapping issues and models. Finally, correctness through cooperative axiomatic reasoning provides an additional basis for understanding parallel algorithm design and specification and is used for run-time assurance of distributed computing systems through operational evaluation.

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

This paper appears, in its entirety, in the Proceedings of the International Summer Institute on Parallel Computer Architectures, Languages, and Algorithms, July 5-10, 1993, Prague, Czech Republic, IEEE Computer Society Press.

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

* 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.

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, $\log_2$ N
             DO in parallel for all $P_j$ where $2^i \leq j \leq N$
                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* $\cdot$ $[\mathbf{a}, \mathit{transpose}(\mathbf{b})]$
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 [14].

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.

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

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

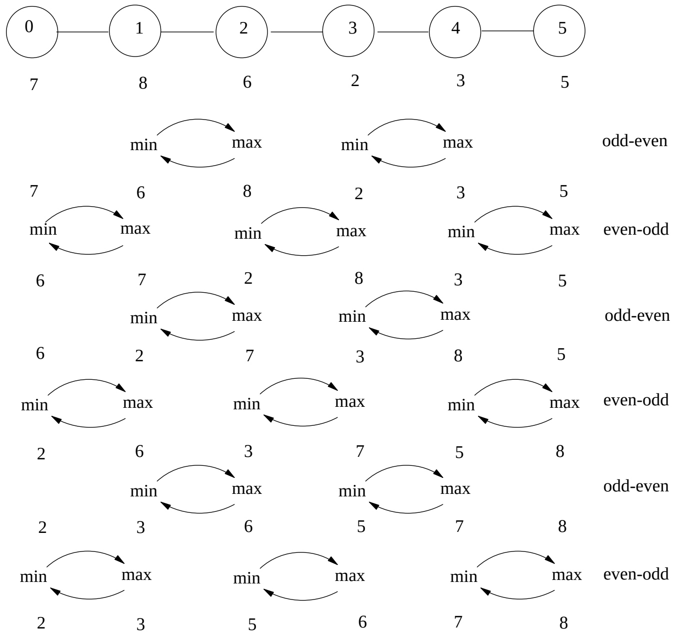


Fig. 1. Odd-Even Transposition Sort

A numerical approximation $\mathbf{U}$ to the solution $\mathbf{u}$ yields the matrix form

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

where the matrix $\mathbf{A}$ 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.

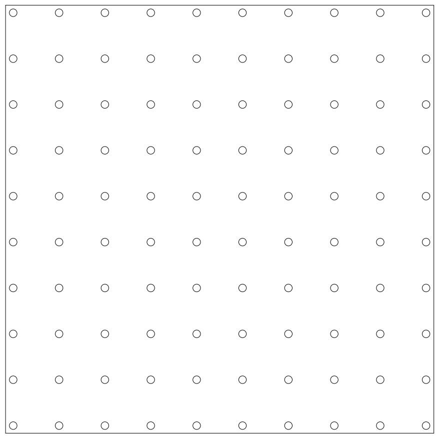


Fig. 2. Discretization of Physical Domain – Domain Decomposition

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

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

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

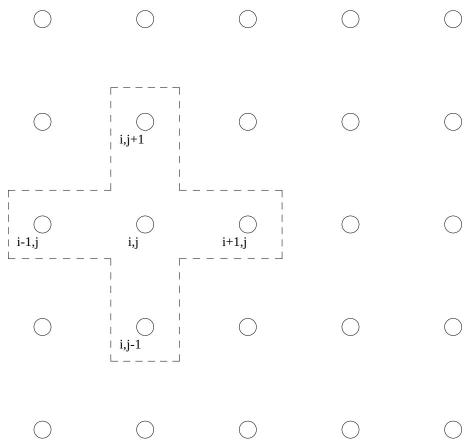


Fig. 3. Localized Computational Molecule

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.

```
$P_i$:
  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.

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.

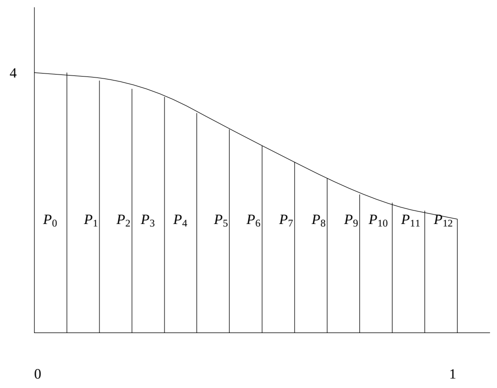


Fig. 4. Domain Decomposition

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.

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

Minsky's conjecture [17] 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

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

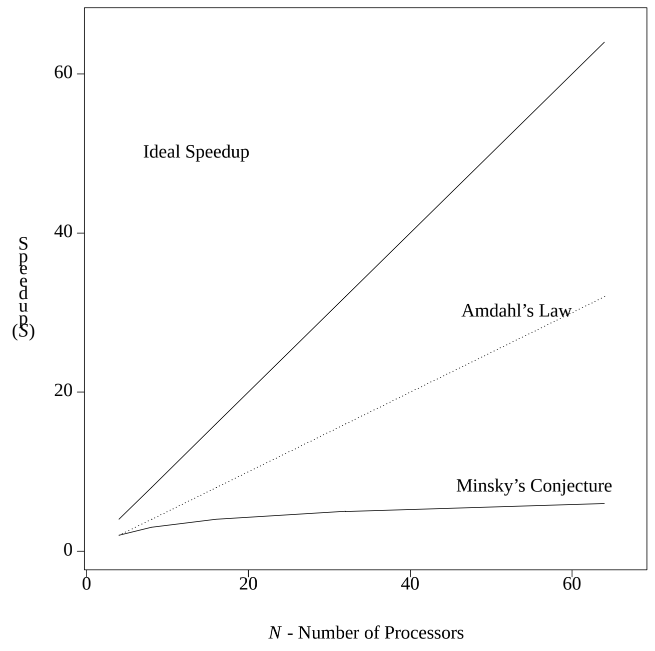


Fig. 5. Speedup Models

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, [12] 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 [12]. 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) [14], 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 [14].

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:

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

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

Alteration and repetition are formed from sets of guarded commands. A guarded command b; c $\rightarrow$ 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 [15], 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

$$M = (Q, I, \Sigma, \delta, \flat, q_0, F)$$

where Q is the finite set of states, Σ is the tape alphabet, $I \subseteq \Sigma$ is the input, δ is the move function, $\flat$ 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 [23] 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 [17] 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 [17], 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.

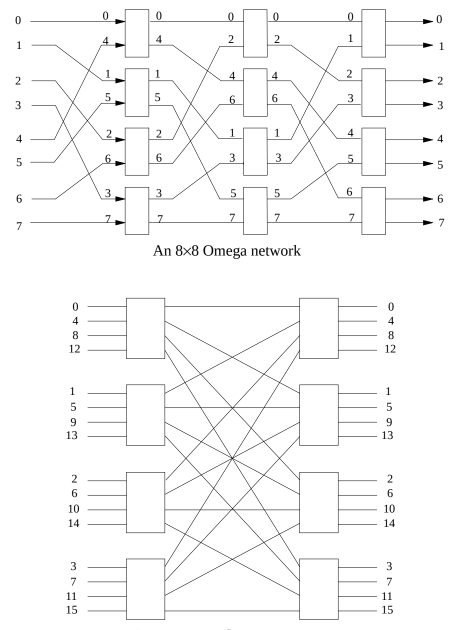
The multistage interconnect is the basis for many commercial and research parallel processors such as PASM [34] and the IBM RS/6000-based POWERparallel 1 (SP1) System [18]. However, if we examine the examples of Section 1 the communication patterns between processors are all *nearest neighbor*. Indeed, 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.

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


An 8×8 Omega network

A 16 node BBN Butterfly interconnection stage

Fig. 6. Sample Multistage Interconnection Networks

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 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 [6]. 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, [36]).

Let f be an embedding function which maps a guest graph G into a host graph H. $|V_G|$ denotes the

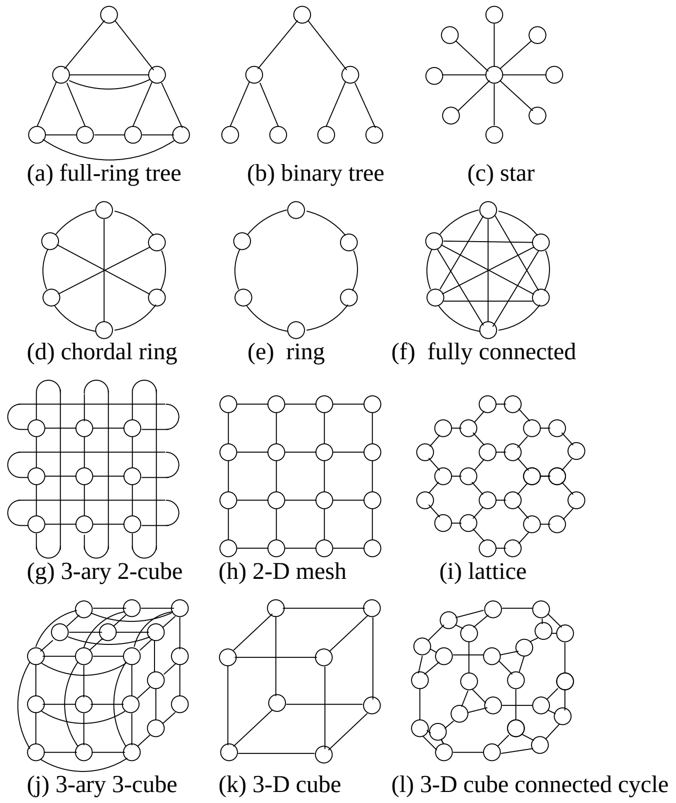


Fig. 7. Some Interconnection Topologies

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

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

where

$$i_b^+ = (i_b + 1) \bmod k$$

and

$$i_b^- = (i_b - 1) \bmod k$$

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.

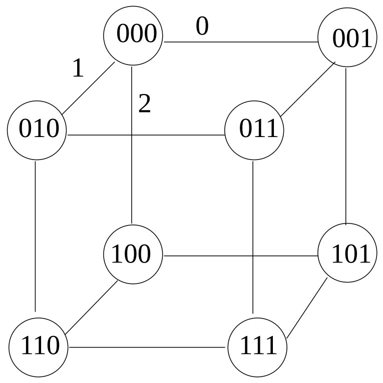


Fig. 8. A 2-ary 3-cube

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.

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

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

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

$$\text{where } 1 \le i_1 < i_2 < \ldots < i_x \le n,\ 1 \le j_1 < j_2 < \ldots < j_x \le m$$

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 [19].

A parallel version of the dynamic programming algorithm is quite straightforward to derive [8]. 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.

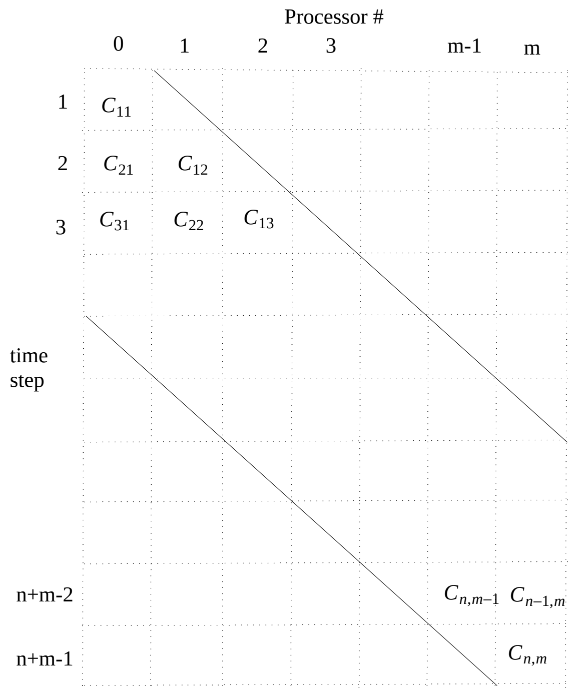


Fig. 9. Diagram indicating activity of processor i at time step t. If $1 \le t-p \le n$, then processor i computes $C_{t-p,p+1}$ at step t. Otherwise, the processor is inactive.

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

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

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

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 $4 \le p \le 2^n$.*

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 [32] 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.

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

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

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:

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

$$\begin{aligned}\zeta^{n+1}_{i,j} = \zeta^n_{i,j} + \Delta t\Bigg[&\frac{-u^n_{i+1,j}\zeta^n_{i+1,j} - u^n_{i-1,j}\zeta^n_{i-1,j}}{2\Delta x} + \frac{-v^n_{i,j+1}\zeta^n_{i,j+1} - v^n_{i,j-1}\zeta^n_{i,j-1}}{2\Delta y}\\ &+\frac{1}{Re}\left(\frac{\zeta^n_{i+1,j} + \zeta^n_{i-1,j} - 2\zeta_i,j^n}{\Delta x^2} + \frac{\zeta^n_{i,j+1} + \zeta^n_{i,j-1} - 2\zeta^n_{i,j}}{\Delta y^2}\right)\Bigg]\end{aligned} \tag{4}$$

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:

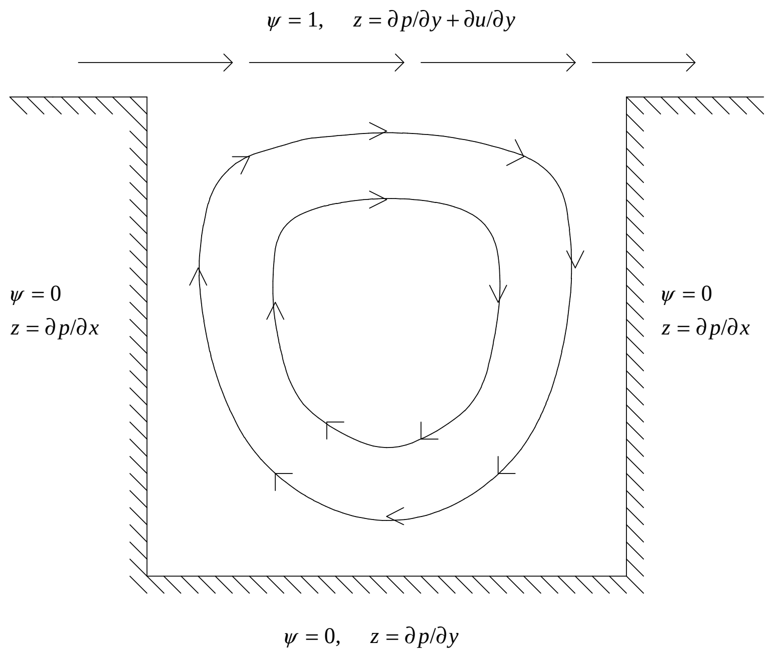

$\psi = 1, \quad z = \partial p/\partial y + \partial u/\partial y$
$\psi = 0$
$z = \partial p/\partial x$
$\psi = 0$
$z = \partial p/\partial x$
$\psi = 0, \quad z = \partial p/\partial y$

Fig. 10. Cavity Driven Flow

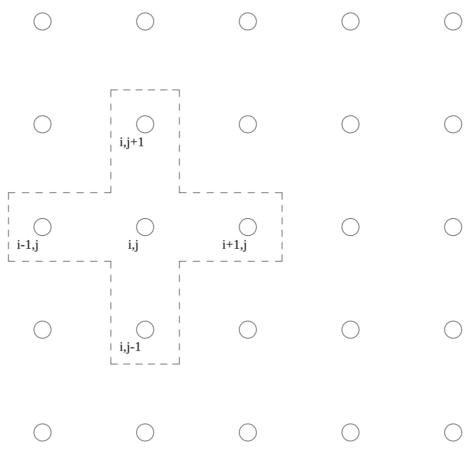


Fig. 11. Localized Computational Molecule

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

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

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 [11] 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, $c_{2,3} \leftarrow a_{2,1}b_{1,3} + a_{2,2}b_{2,3} + a_{2,3}b_{3,3} + a_{2,0}b_{0,3}$

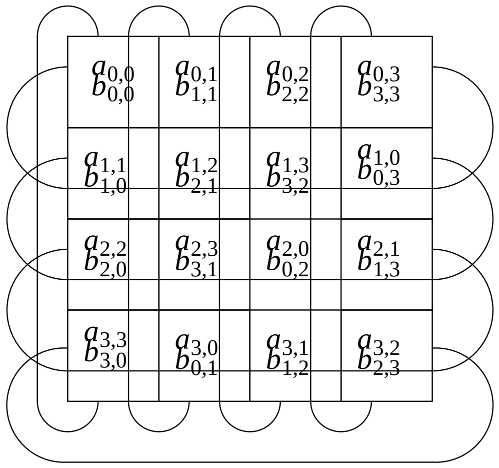


Fig. 12. Toroidal Shift

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,

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

where

$$i^+ = (i+1) \bmod j$$

and

$$i^- = (i-1) \bmod j$$

and their Gray code ordering

$$\begin{array}{cc} G(i_1)G(i_0^+) & G(i_1)G(i_0^-) \\ G(i_1^+)G(i_0) & G(i_1^-)G(i_0) \end{array}$$

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 of $d = 3$ is shown in Figure 13

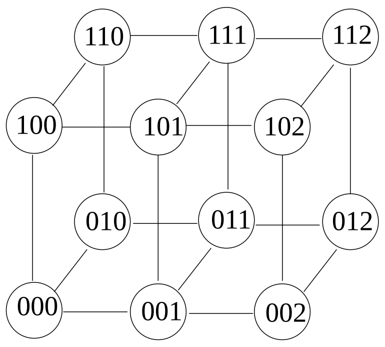


Fig. 13. 3D Mesh Interconnection

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 $G_{k_{d-1}}(i_{d-1}) \times \cdots \times G_{k_1}(i_1) \times G_{k_0}(i_0)$.*

Pyramid Embedding 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 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 this answer to finer and finer arrays as shown in Figure 14. Although this does work, multi-grid methods are much more powerful than this simple concept. Given the system of

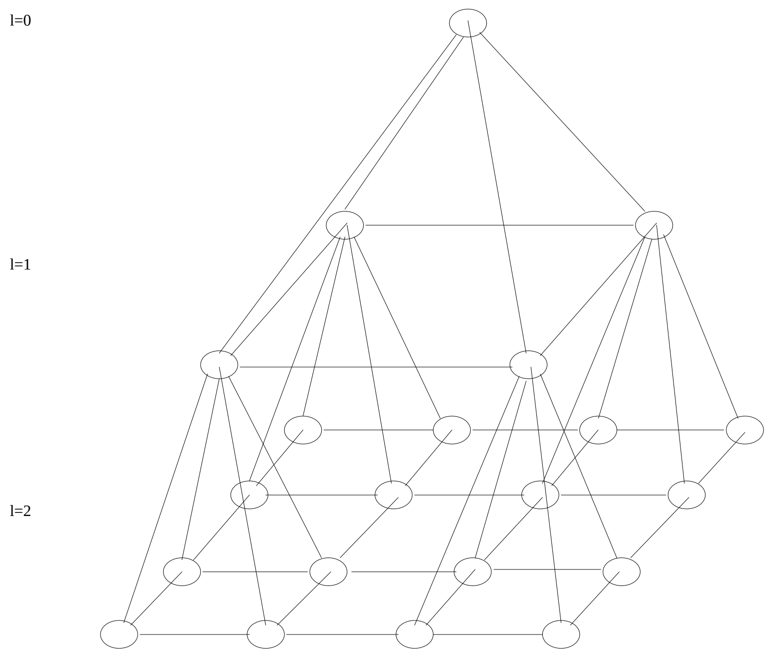


Fig. 14. Multgrid structure for N=16 Processors at the Finest Level

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,

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

and finally,

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

where $\mathbf{R}$ denotes the residual, $\mathbf{R} = \mathbf{F} - \mathbf{AV}$. This resulting equation can be solved exactly as the first equation, since all of the variables except $\mathbf{E}$ 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 15. What may be seen as a relatively smooth change at the finest level appears as rapid changes when restricted to a coarser 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.

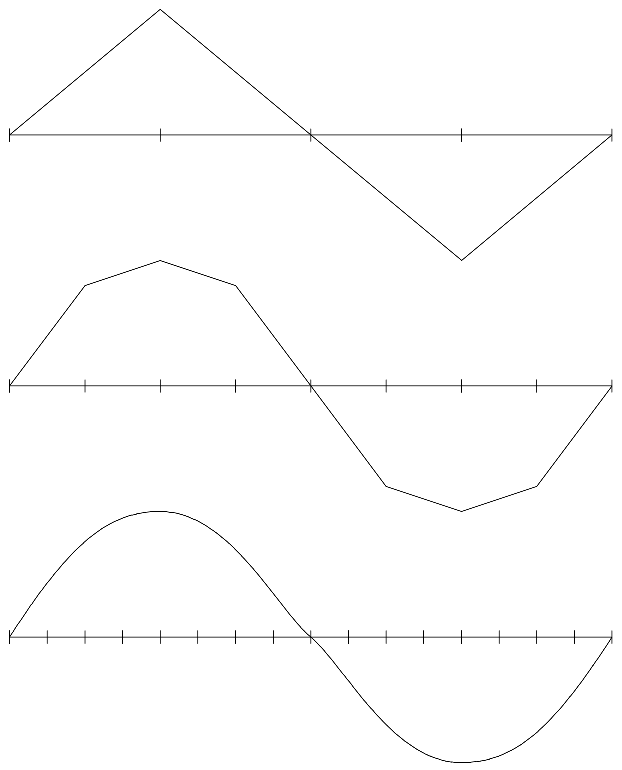


Fig. 15. Error Frequency Reduction Using Multgrid

As illustrated in Figure 16 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.

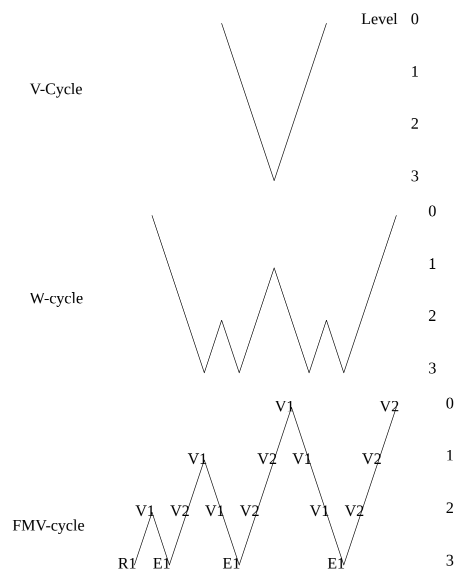


Fig. 16. V and W Cycles of the Multigrid Method

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, 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 14, 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 21. A Hierarchical Binary Reflected Gray Code (HBRGC) is a BRGC such that

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

Definition 22. The Hierarchical Binary Reflected Gray Code HG_k is a code of length k such that

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

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 17.

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

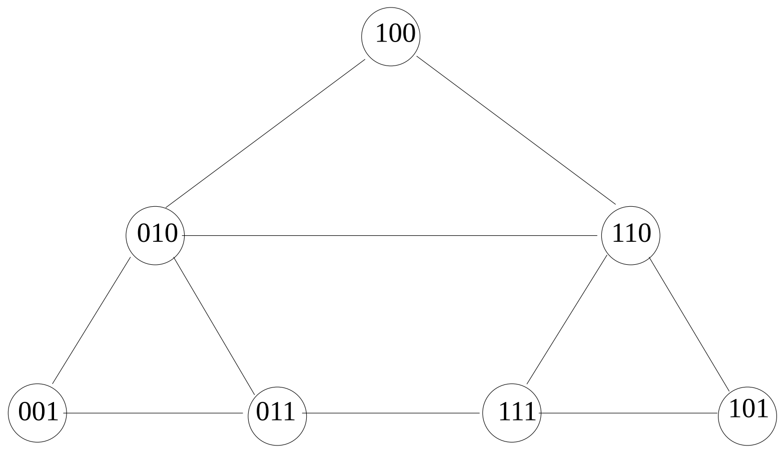


Fig. 17. 2D Pyramid Generated by HBRGC HB

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 [10], 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,

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

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

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 a 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 [9].

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* [13]. 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 [5].

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

$$\begin{array}{l} \rho_1 :: \rho_2 ! x \\ \rho_2 :: \rho_1 ? y ; \rho_3 ! y \\ \rho_3 :: \rho_2 \end{array}$$

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

$$\begin{array}{lll} < x = z > & \rho_1 & < x = z > \\ < true > & \rho_2 & < y = z > \\ < true > & \rho_3 & < u = z > \end{array}$$

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 [21]. 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 [35].

It has been shown [22] 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 [21] 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 [13].

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 $P_{\bar{e}}^{\bar{x}}$[1] is true before the multiple assignment is executed and execution terminates, then P will be true after the assignment. Thus we have the following:

[1] 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.

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

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.

[21] does not have distributed termination which is contrary to Hoare's original version of CSP [14]. 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

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

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

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

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:

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

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

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

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

In order to prove 16, it is sufficient to prove

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

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

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 [7] 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:

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

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

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 [21] 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 [30].

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 [33]. The work of [33] 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.

6 Operational Evaluation

It is important for both life-critical, and non-life-critical distributed systems to meet their specification at run time [20]. Large, complex, distributed systems, are subject to individual component failures which can cause system failure. Fault tolerance is an important technique to improve system reliability. The fault detection aspect identifies individual faulty components (processors) before they can affect, negatively, overall system reliability.

A failure occurs when the user observes that a resource does not perform as expected. The failure is the result of some part of the resource entering a state which is contrary to the specification of the part. The cause of the resource entering such a state is referred to as a fault. When a system can recover from a fault without exhibiting a failure, then the system has fault tolerance. Reliability is a measure of the probability that a specific resource will perform a required function for a specified period of time, usually the item's life time, even in the presence of faults. The higher the probability the higher the reliability of the system is considered to be.

Many methodologies for improving system reliability have been developed throughout the years. These different methodologies fall into two basic groups: fault masking techniques and concurrent techniques. Early attempts at improving system reliability used fault-masking methods; these methods make the hardware tolerant of faults through the multiplicity of processing resources. In contrast, concurrent fault detection methods attempt to locate component errors which can lead to system failure. Once the faults are identified, reconfiguration and recovery [37] are used to deal with the fault. This paper focuses on detecting the occurrence of errors. Recovery and reconfiguration are different issues. Work in concurrent detection methods includes self-checking software [38] and recovery blocks [31], which instrument the software with assertions on the program's state, watchdog processor [28], which monitors intermediate data of a computation, and algorithm-based fault tolerance [16] which imposes an additional structure on the data to detect errors. These methods define structure for fault tolerance, but do not, generally, give a methodology for instantiating the structure.

Application-oriented fault tolerance [29], by contrast, provides a heuristic approach, based on the "Natural Constraints," to choosing executable assertions from the software specification. These executable assertions [38], in the form of source language statements, are inserted into a program for monitoring the run-time execution behavior of the program. The general form is as follows:

$$\textbf{if}\ \neg\, ASSERTION\ \textbf{then}\ ERROR$$

Executable assertions are used to ensure that the program state, in the actual run-time environment, is consistent with the logical state specified in the assertion; if not, then an error has occurred and a reliable communication of this diagnostic information is provided to the system such that reconfiguration and recovery can take place. The heuristics for selection of the actual executable assertions are based on three metrics of *progress*, *feasibility*, and *consistency*.

What our earlier work lacks is a theoretical foundation built upon mathematical models and theories. 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. This paper describes *Changeling* as a formal method using the mathematical model of axiomatic program verification to construct executable assertions for error checking in distributed systems. Application of the *Changeling* system is a two step process. First, from a verification proof outline, *Changeling* converts a shared memory proof outline into a distributed memory proof outline, which closely matches the distributed operational environment. Second, *Changeling* transforms the assertions from the proof outline into executable assertions.

6.1 Changeling and Application-oriented Fault Tolerance

Application-oriented fault tolerance works on the principle of testing at run time the intermediate logical assertions from the verification proof outline i.e. application-oriented fault tolerance works on the following principle:

> *If we test and ensure intermediate results of a program's computation meet its specification, the end solution meets its specification if the intermediate results meet their specification. If processor errors occur that do not affect the solution, then they are not errors of interest. Program verification provides these tests.*

The above principle yields a formal statement of application-oriented fault tolerance; we generate the executable assertions from the logical assertions used in the verification proof outline of $< P > S < Q >$. The executable assertion generated corresponding to any logical assertion Q_i from the verification proof outline is the following:

$$\textbf{if } \neg\, Q_i \textbf{ then } ERROR$$

Formally, this ensures that if P is true before the concurrent program S begins execution, S tests at run time that S satisfies the specification as defined by P and Q, by using the embedded executable assertions generated from the assertions of the verification proof. Conversely, the assertions of the verification proof represent the properties that must be satisfied by the run-time environment; an error that causes the execution of the program not to satisfy the specified assertions will be flagged as an error by the executable assertions.

The reader may be suspicious that some program S may be changed into a program S' by an error that satisfies the specification as defined by P and Q. Consider, as an example, a program S computing some value x with postassertion $< Q > \equiv < x \geq 0 >$. Suppose that S should compute $x = 3$. A program S' may actually compute $x = 4$. The postcondition is still satisfied, although, the value is not what was intended. This is not a problem with the validity of the postassertion, it is a weakness of the specification. If $x = 3$ was what was really intended, then the proper postassertion should have been $< Q > \equiv < x = 3 >$. If $< Q > \equiv < x \geq 0 >$ is a sufficient specification for the application at hand, then there is no problem.

To eliminate confusion between the testing of intermediate results (via logical assertions) for correctness with respect to the algorithm and the evaluation of the executable assertions derived from the verification proof in the run-time environment, we will refer to the former as the *verification environment* and the latter as the *(distributed) operational environment.*

To summarize, the transformation of an algorithm to an error-detecting algorithm involves using the assertions of the verification proof as executable assertions that are to be embedded into the algorithm.

Taking an application from the verification environment to the distributed operational environment is not a straightforward task. It is this difficulty that inspired the development of *Changeling*. *Changeling* consists of four distinct components:

1. The GAA Proof System described in Section 5
2. An HAA proof system which mimics closely the distributed operational environment
3. Formal conversion from GAA to HAA
4. Formal translation of assertions in the HAA proof system to executable assertions and reducing state information to improve run-time efficiency

These components are described in the following paragraphs.

History of Auxiliary Variable (HAA) Verification System The logical assertions from the GAA verification environment cannot be directly used as executable assertions in the distributed environment; in the distributed environment, there are no global variables. Thus, to evaluate, at run time, logical assertions containing global auxiliary variables, an explicit updating mechanism must be created. Here we develop the verification proof system (HAA) in which updates of global auxiliary variables are exchanged at communication time. This matches, more closely, the operational environment. We show that every verification proof outline in the GAA proof system has the same properties in the HAA proof system, i.e., satisfaction and non-interference; thus, implying that the HAA proof system has the soundness and completeness properties of the original GAA proof system. The existence of the HAA proof system allows for proofs that can be directly transformed to executable assertions in the run-time environment.

Developing the HAA system requires us to keep track of which processes communicate with which other processes. Each process needs to record its global auxiliary variable updates with respect to all other processes. When communication occurs between two processes, they need to exchange the updates and locally apply them (the updates). This is formalized in the following definitions.

Definition 23. For a process ρ_i, h_i denotes the sequence of all communications that process ρ_i has so far participated in as the receiving process. Thus, h_i is a list consisting of tuples (these are different from the [35] tuples; all future reference to tuples will refer to the following tuples) representing matching communication pairs of the form

$$[\rho, (Var, Val), T, C]$$

where ρ is a process from which ρ_i receives from, Var is the variable that ρ is transmitting to ρ_i with formal parameter Val. T denotes the time at which the value Val was assigned to variable Var and C denotes the communication path.

Since we have several processes running in parallel and there exists no concept of a global time, the time T is a local time represented by an instantiation counter that is incremented by one after every execution of a statement. This permits an ordering (time-stamping) for all updates of the GAVs within each process.

To be able to account for the different operations performed on the auxiliary variables, each process has to keep a history of variable updates with respect to the last communication with the other processes. These variable sets are described using the subscript of the corresponding process.

Definition 24. Let g_{ij} depict the GAV set in process ρ_i with respect to process ρ_j, i.e., g_{ij} contains the changes that were made to the GAVs in ρ_i since the last communication with ρ_j. G_i is the set of sets $g_{i0}, g_{i1}, \ldots, g_{i(N-1)}$ in process ρ_i. Thus, when two processes ρ_i and ρ_j communicate, the values of their respective subsets, $g_{ij} \in G_i$ and $g_{ji} \in G_j$, are exchanged.

When two processes ρ_i and ρ_j communicate, where ρ_j is the sender, ρ_j will augment the communication by sending the values of global auxiliary variables that ρ_j updated, or received updates of, between the last and current communications between ρ_i and ρ_j. We batch the changes made to the local copies of the global auxiliary variables by ρ_j since the last communication (with any other processor) in g_{jj}. Before a communication, the function ψ applies changes to all g_{jk}'s and g_{jj} is reset to null to collect future changes. Definition 26 formally describes the communication of g_{ji}. Definition 27 formally describes how process ρ_i updates G_i based on the communicated g_{ji} after communication has taken place.

Definition 25. The actual set of GAVs to be sent during a communication between ρ_i and ρ_j, where ρ_j is the sender, is determined based on the variables in g_{jj}, i.e., all the variables that were updated in ρ_j since the last communication with any process. The set g_{jj} is updated every time an assignment to a GAV takes place in ρ_j and reset at communication time to the empty set. The following function $\psi(G_j, g_{jj})$ describes the update of all variable histories before a communication.

$$\begin{aligned}\rho_j :\quad & (\forall k, 0 \leq k \leq N-1)(\forall g_{jk} \in G_j)\\ & [if\ k \neq j\ then\ g_{jk} \leftarrow g_{jk} \cup g_{jj}\ else\ if\ k = j\ then\ g_{jj} \leftarrow \emptyset]\end{aligned}$$

The following definition formally defines the semantics of global auxiliary variable communication.

Definition 26. The *primary* communication is a matching communication pair for the exchange of variables between processes which are not GAVs. It can be described by a tuple $[\rho_j, (Var, Val), t, j]$ where $t = T_j$ is the current value of the local time. It is easy to see that all communications in the GAA system are primary since GAVs are updated globally. An *augmented* communication permits the exchange of the GAVs after a primary communication occurs. In an augmented communication, the values of g_{ji} are marshalled into a message sent to process ρ_i.

For each process ρ_i after an augmented exchange with ρ_j, ρ_i updates its set of GAVs in G_i with the new values received. This interchange is described in Figure 18 for two processes ρ_i, ρ_j and one matching communication pair within the execution sequence of the two processes.

```
For process $P_i$:
/* execute arbitrary set of statements excluding communication but
including assignments to auxiliary variables */
$S_{i1}$; $< T_i := 1 >$
$S_{i2}$; $< T_i := T_i + 1 >$
...;
$S_{ik}$; $< T_i := T_i + j >$
/* update the auxiliary variables */
$G_i \leftarrow \psi(G_i, g_{ii})$; $< T_i := T_i + 1 >$
/* perform communication with process $P_j$;
the first communication represents the actual communication */
/* the next two communications represent the exchange augment of the auxiliary variables */
$P_j$ ? $V$ ; $< T_i := k >$
$P_j$ ? $g_{ji}$ ; $< T_i := k + 1 >$
$P_j$ ! $g_{ij}$ ; $< T_i := k + 2 >$
/* update the auxiliary variables */
$G_i \leftarrow \phi(G_i, g_{ji})$; $< T_i := k + 3 >$

For process $P_j$:
/* execute arbitrary set of statements excluding communication
but including assignments to auxiliary variables */
$S_{j1}$; $< T_j := 1 >$
$S_{j2}$; $< T_j := T_j + 1 >$
...;
$S_{jk}$; $< T_j := T_j + 1 >$
/* update the auxiliary variables */
$G_j \leftarrow \psi(G_j, g_{jj})$; $< T_j := T_j + 1 >$
/* perform communication with process $P_i$;
the first communication represents the actual communication */
/* the next two communications represent the exchange augment of the auxiliary variables */
$P_i$ ! $V$ ; $< T_j := k >$
$P_i$ ! $g_{ji}$ ; $< T_j := k + 1 >$
$P_i$ ? $g_{ij}$ ; $< T_j := k + 2 >$
/* update the auxiliary variables */
$G_j \leftarrow \phi(G_j, g_{ij})$; $< T_j := k + 3 >$
```

Fig. 18. An HAA proof outline for one matching communication pair.

Definition 27. The updates performed in the different processes are described by a function $\phi(G_i, g_{ji})$ on the set of the GAV history and the variables to be updated. The actual update function ϕ is now defined on all the subsets within G_i on tuples of the form $[\rho_j, (g_{ji}, gvar_j), T, j]$.

$$\begin{aligned} \rho_i :\ & (\forall k, 0 \le k \le N-1)(\forall g_{ik} \in G_i) \\ & [if\ k \ne j\ then\ g_{ik} \leftarrow g_{ik} \cup gvar_j \\ & \qquad else\ if\ k = j\ then\ apply(gvar_j); g_{ij} \leftarrow \emptyset\,] \end{aligned}$$

When processes ρ_i and ρ_j communicate, all old values in the set g_{ij} will be replaced by the new variables. Additionally, these new values (from $gvar_j$) are unmarshalled and applied to update the local values of process ρ_i. In this way, communication propagates GAV updates throughout the concurrent program.

It can be seen that the so-called "global auxiliary variables" in the HAA system are not really global in the sense that all processes have the same values of the variables at all times. Indeed, it is likely that at the end of the process execution some processes that ran in parallel will have different values within their set of GAVs. We show that because of non-interference, this is not a problem with respect to the proof system.

Within a process execution, two communicating processes can have arbitrary interleavings of their statements up to the communication, but are conceptually synchronized at the communication point. The assertions will not interfere with each other due to the non-interference property of the GAA system which provides for arbitrary execution orders. Since two (or more) processes will only change (write onto) the same global auxiliary variable if they have to communicate with each other, they will also exchange other variables/data in that process and the values of the auxiliary variables will be available for the other process at the critical point: right after a communication takes place. Thus, sending only the history of the global variable updates instead of immediately providing the other process(es) with the latest information will not cause any problems, since the values of the variables will be available at the communication points, where they are in fact provided.

An example of three possible process execution sequences that are subject to non-interference is shown in Figure 19. For any two processes, non-interference will guarantee that the execution order of the two processes or any arbitrary interleaving of them will not invalidate the assertions made on the respective process statements.

Non-interference and the rule of satisfaction can be used to show that the soundness and completeness properties of the original GAA system will hold in the new HAA proof system.

Theorem 28. [26] *The history of auxiliary variables approach (HAA) retains the properties of the global auxiliary variables approach (GAA).*

Reliable Communication of State Information The HAA proof system provides for direct transformation of assertions from the verification environment into executable assertions for the non-faulty distributed operational environment. However, we are concerned with the distributed faulty environment. Thus, it is necessary to ensure that faulty processors cannot fool executable assertions by incorrect augmented communication of $g's$ through sending inconsistent messages to different processors. It is necessary for this to be detected. This is the purpose of *consistency* executable assertions. Mathematically, this can be described as follows:

Definition 29. For a non-faulty process ρ_i, if there exist any two tuples $t_1,\ t_2 \in h_i$ such that

$$t_1 = [j, (Var, Val_1), T, C_1]$$

$$t_2 = [j, (Var, Val_2), T, C_2]$$

then if $Val_1 \circ Val_2$ the system is said to be inconsistent otherwise the system is said to be consistent.

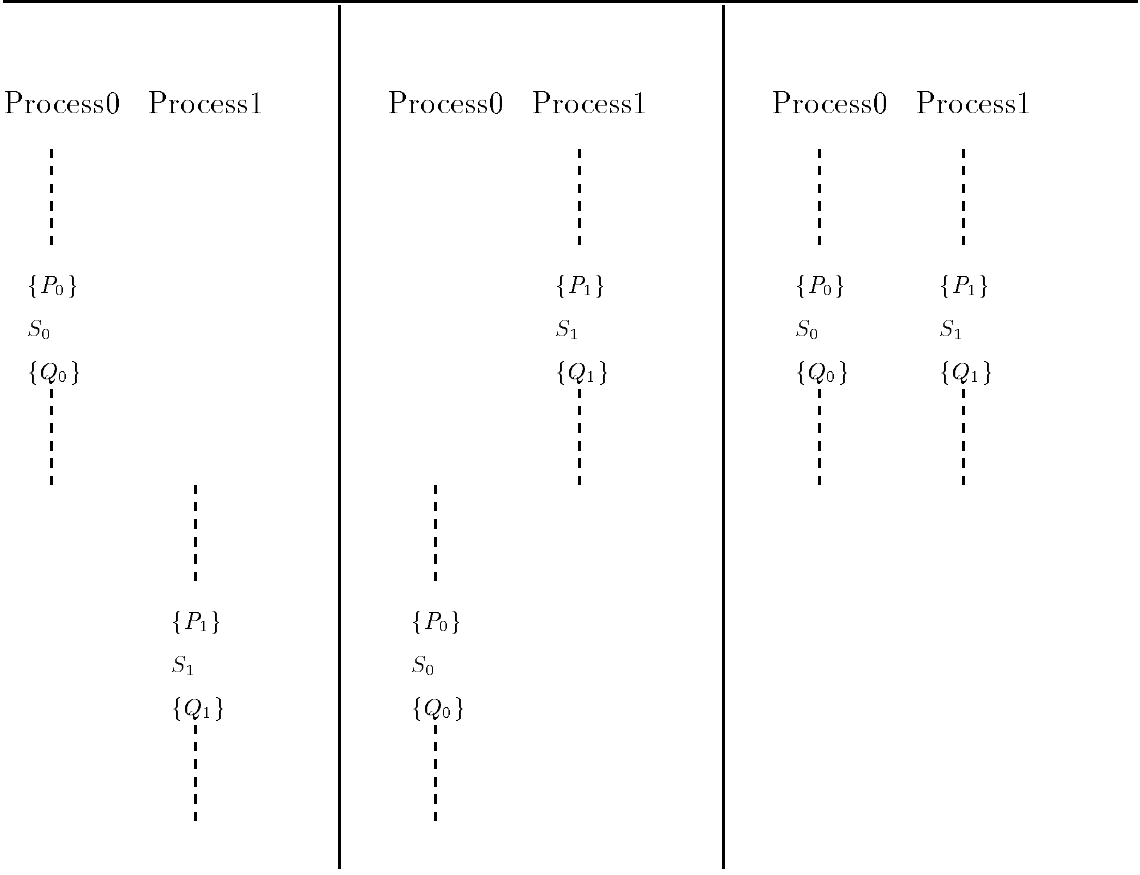


Fig. 19. Some possible process execution sequences before communication takes place.

$\circ$ is defined as a set of functions such that each $\circ' \in \circ$ is of functionality $dt \rightarrow T, F$ where dt is an abstract data type. Examples of $\circ'$ are $\neq$, $\subseteq$, $\neg prefix$, or some other operator appropriate to the choice of the data type of Var. Where no ambiguity results, we will refer to a particular $\circ'$ simply as $\circ$.

The strongest motivation for the consistency condition is to supplement the power of the executable assertions derived from the HAA system. When the value of a variable computed in time T is communicated to a set of processors on more than one path, there will be two or more tuples in h_i that satisfy the precondition. Under a bounded number of faults, the consistency definition of 29 ensures that a non-faulty processor receives a consistent set of input values for its executable assertions, otherwise, $Val_1 \circ Val_2$, and an inconsistent system can be detected. The degree of fault tolerance is based on standard network flow arguments and is not repeated here. It should be noted that all faults in communication links are mapped to a processor, thus it is enough to assume only faulty processors.

Consistency does not have to be explicit. In other words, an error-detecting program may have to explicitly add code to implement consistency. This can be done in many ways. There are classes of problems that have the property of natural redundancy in the problem variables. This implies that there are types of errors, which if they occur at stage i, eventually, at some stage j (where $j > i$), we have that stage j satisfies the properties as defined by the intermediate assertions of a verification proof, despite the fact that the error had occurred in stage i. If a program variable is naturally redundant then this means that this program variable can be constructed from other variables.

Run-Time Efficiency Considerations The transformation from the HAA verification environment to the operational environment described above is optimal in the sense that all violations of the program's specification (in terms of the postconditions on each statement and within the limits of consistency) are caught under a bounded number of faults. However, when run-time efficiency is considered, not all of these assertions, nor all of the communicated GAVs are necessary. These two aspects of reducing complexity are treated as follows:

– Assertions involving *local variables* to a particular process which are necessary in the verification environment are useless in the distributed operational environment. Since the unit of failure and reconfiguration is at the processor level, a processor cannot be trusted to diagnose itself as faulty or fault-free. Thus, assertions using only local variables incur a run-time overhead that is not necessary and all such assertions can be deleted.
– The fault coverage of certain assertions using the GAVs may be *subsumed*. Thus, many of the remaining assertions may be removed as well. Likewise, removing some of the assertions may result in certain GAVs no longer being required. Furthermore, certain assertions may be too expensive to evaluate in the operational environment and may be deleted for that reason.

We applied this transformation to several concurrent applications including concurrent database transactions schedules [24], bitonic sorting [25], and concurrent branch and bound [27] and obtained performance and error coverage data on each.

7 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. M. Clint. Program proving: coroutines,. *Acta Informatica*, 2:50–63, 1973.
6. G. Cybenko, D. W. Krumme, and K. N. Venkataraman. Fixed hypercube embedding. *Information Processing Letters*, 25:35–39, 1987.
7. E. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Inc., 1976.
8. 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.
9. 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.
10. 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.
11. W. M. Gentleman. Some complexity results for matrix computations on parallel computers. *Journal of the ACM*, 25(1):112–115, January 1978.
12. J. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.
13. C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

14. C. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.
15. J. Hong. *Computation, Computability, Similarity and Duality.* Pittman, London, 1986.
16. K Huang and J. Abraham. Fault-tolerant algorithms and their applications to solving Laplace equations. *Proceedings of the 1984 International Conference on Parallel Processing*, pages 117–122, August, 1984.
17. K. Hwang and Briggs F. *Computer Architecture and Parallel Processing.* McGraw-Hill, New York, 1984.
18. IBM. *IBM Scalable PowerParallel System 9076-SP1*, 1993.
19. 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.
20. J Laprie and B. Littlewood. Probabilistic assessment of safety-critical software: Why and how? *Communications of the ACM*, 35(2):13–21, 1992.
21. G.M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15:281–302, 1981.
22. H. Lutfiyya and B. McMillin. Comparison of three axiomatic proof systems. *UMR Department of Computer Science Technical Report CSC91-13*, 1991.
23. 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.
24. H. Lutfiyya, B. McMillin, and Alan Su. Formal derivation of an error-detecting distributed data scheduler using CHANGELING. In *Formal Methods in Programming*, Novosibirisk, Russia, July 1993. Also as UMR Department of Computer Science Technical Report Number CSC 92-14.
25. H. Lutfiyya, M. Schollmeyer, and B. McMillin. Fault-tolerant distributed sort generated from a verification proof outline. In H. Kopetz and Y. Kakuda, editors, *Responsive Computer Systems - Dependable Computing and Fault-Tolerance*, volume 7. Springer-Verlag, 1992. Also as a Short Talk in the 14th ICSE, Melbourne, Australia and UMR Department of Computer Science Technical Report C.Sc. 91-12.
26. H. Lutfiyya, M. Schollmeyer, and B. McMillin. Formal generation of executable assertions for application-oriented fault tolerance. *UMR Department of Computer Science Technical Report Number CSC 92-15*, 1992.
27. H. Lutfiyya, A. Sun, and B. McMillin. A fault tolerant branch and bound algorithm derived from program verification. *IEEE Computers Software and Applications Conference(COMPSAC)*, pages 182–187, 1992.
28. A. Mahmood, E. McCluskey, and D. Lu. Concurrent fault detection using a watchdog processor and assertions. *IEEE 1983 International Test Conference*, pages 622–628, 1983.
29. B. McMillin and L. Ni. Reliable distributed sorting through the application-oriented fault tolerance paradigm. *IEEE Trans. of Parallel and Distributed Computing*, 3(4):411–420, 1992.
30. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.
31. B. Randall. System structure for software fault tolerance. *IEEE Transactions of Software Engineering*, SE-1(2):220–232, 1975.
32. 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.
33. 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.
34. H.J. Siegel et al. PASM: A partionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Transactions on Computers*, C-30:934–947, December 1981.
35. N. Soundararahan. Axiomatic semantics of communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 6(6):647–662, 1984.
36. Q. F. Stout. Hypercubes and pyramids. In V. Cantoni and S. Levialdi, editors, *Pyramidal Systems for Computer Vision.* Springer-Verlag, New York, 1986.
37. R. Yanney and J. Hayes. Distributed recovery in fault tolerance multiprocessor networks. *4th International Conference on Distributed Computing Systems*, pages 514–525, 1984.
38. S. Yau and R. Cheung. Design of self-checking software. *Proc. Int'l Conf. on Reliability Software*, pages 450–457, April 1975.