1982

A Model for Analyzing Generalized Interprocessor Communication Systems

Janice D. Cuny

Lawrence Synder

Report Number:
81-406

TR #406

A MODEL FOR ANALYZING GENERALIZED INTERPROCESSOR COMMUNICATION SYSTEMS

Janice E. Cuny
Lawrence Snyder

Computer Sciences Department,
Purdue University
West Lafayette, Indiana

ABSTRACT

We present a model of parallel computation which is parameterized by execution mode, allowing us to express different modes within a common framework. The model enables us to make legitimate comparisons of execution modes. We report here on some preliminary results.

October 6, 1982

This work is part of the Blue CHiP Project. It is supported in part by the Office of Naval Research Contracts N00014-80-K-0816 and N00014-81-K-0360. The latter is Task SRO100.

A MODEL FOR ANALYZING GENERALIZED INTERPROCESSOR COMMUNICATION SYSTEMS

Janice E. Cuny

Lawrence Snyder

Computer Sciences Department,

Purdue University

West Lafayette, Indiana

Algorithmically-specialized computers are likely to be parallel machines since parallelism is an effective method of circumventing the physical limits of switching and signal transmission delays. If so, then an important design decision is whether the algorithmically-specialized computer executes synchronously, asynchronously or in an intermediate mode such as data-driven execution. The decision is crucial because it influences cost, performance and the convenience of programming. For asynchronous execution, there is overhead associated with processor to processor communication because of the requisite hand-shaking protocol. Data-driven execution must be charged for the additional circuitry needed to buffer data arriving at a processor prior to its use and to provide a signalling-back mechanism indicating when buffer space is available. To their credit, both mechanisms appear to be easy to program,

although the programs are subject to possible deadlocks. Synchronous execution has none of the overhead problems, nor is it subject to deadlock. However, assuming (as is reasonable) that the single "steps" of an abstract algorithm are implemented by varying numbers of more primitive processor steps, idles will have to be inserted in some processors so that they match the execution rate of the processors with which they communicate. There are cases where this cannot be done. Moreover, when it can be done, the resulting programs can be problem-size dependent, hardware dependent, and extremely difficult to write. These are important considerations that cannot be easily dismissed.

In order to evaluate the consequences of problems such as these, we have developed a model for analyzing general interprocessor communication. What makes the model unique and especially useful for the problems mentioned above is that it is *parameterized by execution mode*. This enables different execution modes to be expressed in one formalism in which fair and accurate comparisons can be made.

The purpose of the paper is to present the model in its full generality and to summarize our early experience with it.

A MODEL OF PARALLEL PROGRAMS

We assume that a parallel processor is composed of m processing elements $M_1, M_2, \ldots, M_m$ which collectively implement an algorithm. The processing elements (PEs) have local memories for program and data storage, and they execute sequential programs under the control of their own program counters. We are concerned only with the input/output behavior of these machines. To avoid hiding communication costs, we assume that the PEs do not share any common memory; instead they communicate through read and write operations. On each time step, a PE can attempt a set of I/O operations simultaneously. Whether or not an operation executes when it is attempted depends on the execution mode. An operation that does not execute is retried on the next step and a process does not proceed with a new set of operations until all of its current operations have completed.

We model such systems as *Interprocess Communication (IC) Systems*. An IC system is completely defined by a function, Λ (mnemonic for "advance"), giving the execution mode of the system and a set of sequences $V_1, V_2, \ldots, V_m$, each describing the behavior of a single PE. The i-th sequence describes the behavior of the i-th machine. There are three types of operations which are represented as follows

reads: the read of value σ from PE i is denoted $r_{i,\sigma}$;

writes: the write of value σ to PE i is denoted $w_{i,\sigma}$;

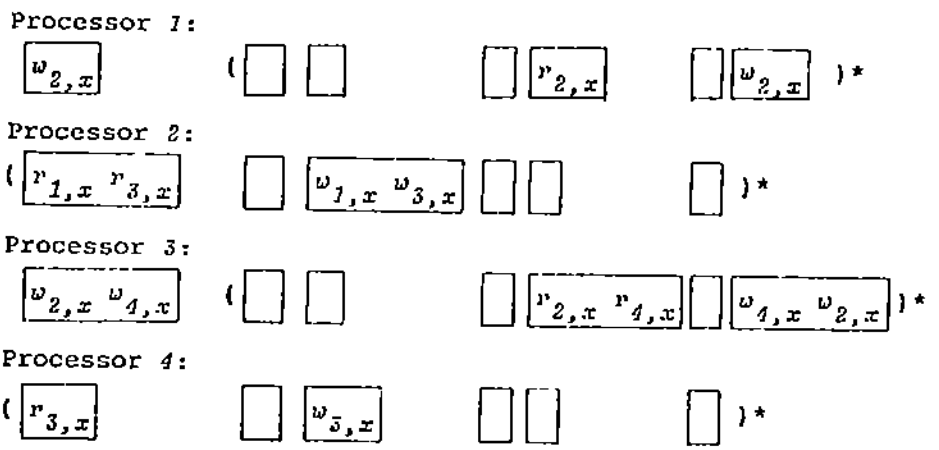
and

time delays: a delay of n time units is denoted d_n (these delays are used only in asynchronous mode as described below).

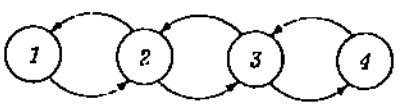
Each symbol in a behavioral sequence is a (possibly empty) set of these operations subject to two restrictions: there is at most one time delay operation in any set (if there is no time delay, the operation is assumed to require one time step); and there is not more than one read (write) to (from) any PE in a single set. Figure 1(a) is an IC system representing the systolic processor for band matrix-vector multiplication with a bandwidth of four [6]. † The sequences of operation sets for each PE are specified by regular expressions. Since the system is synchronous, there are no time delay operations and since the system does not have data-dependent branches, we represent the transmitted values by a single, generic value x. Figure 1(b) shows the *communication graph* for this system; each vertex represents a PE and a directed edge from node i to node j represents a *communication link* over which the i-th PE writes to the j-th PE and the j-th PE reads from the i-th PE.

---

† Note that in our figures we use rectangular boxes to enclose sets rather than the usual brace notation.



1(a) IC system representing systolic processor for band matrix - vector multiplication.



1 (b) Communication graph for the IC system of Figure 1(a).

Figure 1.

We define the execution of an IC system in terms of three sequences, $C^1,C^2,C^3,\ldots$, $\Delta^1,\Delta^2,\Delta^3,\ldots$, and $Q^1,Q^2,Q^3,\ldots$. For all $k>0$, C^k describes the set of operations that are attempted on the k-th execution step, Δ^k describes the time needed for those operations to complete, and Q^k describes the status of communications if they all do complete. Each element of the first sequence is an m-vector giving program counter values (indexes into operation set sequences) for all PEs. Each element of the second sequence is an m-vector giving timer values (the number of steps that

must elapse before the completion of the current operation set) for all PEs. Each element of the third sequence is an $m \times m$ matrix of strings, giving the status of communications in terms of strings of messages and requests. The status of communications on the link from PE i to PE j is given by $q_{i,j}$. Values that have been written but that have not yet been read are denoted by elements of an alphabet Σ; values that have been requested but that have not yet been written are denoted by their inverses.† $q_{i,j}$ is a queue of written values (head on the right end) followed by a queue of requested values (head on the left end); corresponding writes and reads cancel at the boundary between these queues.

To start the sequences we define, for all $i,j \in [m]$, †† $c_i^1=1$ and

$$\delta_i^1 = \begin{cases} n & \text{if } d_n \in V(c_i^1) \\ 1 & \text{otherwise} \end{cases}$$

and $q_{i,j}^1 = a \cdot b$ where

$$a = \begin{cases} \sigma & \text{if } w_{j,\sigma} \in V_i(c_i^1) \\ \lambda & \text{otherwise} \end{cases}$$

and

$$b = \begin{cases} \sigma^{-1} & \text{if } r_{i,\sigma} \in V_j(c_j^1) \\ \lambda & \text{otherwise} \end{cases}$$

---

† We represent the inverse of a symbol σ by σ^{-1} and define $\sigma \cdot \sigma^{-1}$ to be λ, the empty string; $\Sigma^{-1} = \{\sigma^{-1} | \sigma \in \Sigma\}$

†† $[m]$ denotes the set $\{1,2,3,\ldots,m\}$.

with $V(j)$ denoting the j-th set of operations in the sequence V.

C^1 shows all PEs executing their first set of operations, Δ^1 shows all of the timer values set to their initial values, and Q^1 shows that the initial reads and writes are pending. The remainder of the sequence of Cs is defined to reflect the fact that a PE moves to a new set of operations only if all operations in its previous set have completed: for $k>0$

$$c_i^{k+1} = \begin{cases} c_i^k+1 & \text{if } A(i,k) \\ c_i^k & \text{otherwise} \end{cases}$$

where $A(i,k)$ is true if the i-th PE finishes the c_i^k-th operation set in step k. The exact form of A depends on the mode of execution and is discussed below. For $k>0$, Δ^k is defined so that the timers are set by the execution of a d operation (default $d=1$) and are decremented by 1 on each subsequent step until they reach 0:

$$\delta_i^{k+1} = \begin{cases} n & \text{if } d_n \in V(c_i^{k+1}) \wedge c_i^k \neq c_i^{k+1} \\ 1 & \text{if } d_n \notin V(c_i^{k+1}) \wedge c_i^k \neq c_i^{k+1} \\ \delta_i^k - 1 & \text{otherwise} \end{cases} .$$

The remainder of the sequence of Qs is defined to reflect the execution of read and write operations: for $k>0$

$q_{i,j}^{k+1} = a \cdot q_{i,j}^k \cdot b$ where

$$a = \begin{cases} \sigma & \text{if } w_{j,\sigma} \subset V_i(c_i^{k+1}) \wedge c_i^{k+1} \neq c_i^k \\ \lambda & \text{otherwise} \end{cases}$$

and

$$b = \begin{cases} \sigma^{-1} & \text{if } \tau_{i,\sigma} \in V_j(c_j^{k+1}) \wedge c_j^{k+1} \neq c_j^k \\ \lambda & \text{otherwise} \end{cases} .$$

We observe that our execution rules are more general and more realistic than those used in many models because we do not insist that all of the operations in a set execute simultaneously. Depending on the definition of A, it is possible, for example, to allow independent reading and writing on different ports.

The execution of an IC system is parameterized by the predicate A. All of the common forms of execution modes can be succinctly expressed within our model:

Execution Mode	$A(i,k)$
synchronous	$\forall i,j \in [m]\ (q_{i,j}^k = \lambda)$
data-driven (unbounded buffer)	$\forall j \in [m]\ (q_{j,i}^k \in \Sigma^*)$
data-driven (b-bounded buffer)	$\forall j \in [m]\ (q_{j,i}^k \in \Sigma^b)$
data-driven (lazy evaluation)	$\exists j \in [m]\ (q_{i,j}^k \in \Sigma^{-1}) \wedge$ $\forall j \in [m]\ (q_{j,i}^k \in \Sigma^*)$
asynchronous	$\delta_i^k = 1$

Parameterizing our model with the execution mode enables us to compare modes and it distinguishes our model from previous formal models

of computation such as the model proposed by Lipton, Miller and Snyder [7], Petri Nets [8], the vector addition system model [5] and the model developed by Arjomandi, Fischer and Lynch [1].

PRELIMINARY RESULTS

Our initial work has been practically motivated: we would like to be able to program algorithms for the CHiP machine [9]. In particular, we are working with an architecture in which computational operations are executed synchronously but I/O operations may be either asynchronous or synchronous. In asynchronous mode, a read that occurs before the corresponding write is delayed and a write that occurs before the corresponding read interrupts the destination PE to buffer the transmitted value. In synchronous mode, I/O interrupts are masked off and corresponding reads and writes must occur simultaneously. A program that can be run fully in synchronous mode is said to be *coordinated*.

We would like, whenever possible, to run coordinated programs. Unfortunately it is extremely difficult for programmers to produce such code and the the code itself is often problem-size and hardware dependent. To surmount these problems, we have developed and implemented algorithms for the automatic synthesis of coordinated programs from data-driven programs [2]. These algorithms enable the programmer to work in the more natural data-driven environment without forfeiting any

of the advantages of a synchronous device. They apply only to *loop programs* in which each PE executes a single loop of instructions. (This restriction at first may seem quite prohibitive but, in fact, most recent algorithms for algorithmically-specialized computers are loop programs. In addition, many programs can be viewed as loop programs by collapsing parallel branches that have the same input/output behavior.)

We have developed two synthesis algorithms. The first, the Wave Algorithm, works on all data-driven loop programs for which conversion is possible but in some cases it produces inefficient code. The second algorithm, the Buffered Write Algorithm, works for only a subset of loop programs and, although it often increases the length of PE code significantly, the results are more efficient. We are currently working to expand the class of programs that we can convert and to develop measures for accurately evaluating the efficiency and trade-offs of our algorithms.

For the programs that we cannot coordinate or that, for reasons of efficiency, require manual design, we have developed and implemented algorithms for testing coordination [3]. We have efficient algorithms for testing the coordination of loop programs and the worst-case coordination of arbitrary programs. The general testing question is PSPACE-hard [4]. We expect that as libraries of parallel programs become available, our algorithms will be useful in determining their interface compatibilities.

REFERENCES

[1] E. Arjomandi, M. Fischer, and N. Lynch, A difference in efficiency between synchronous and asynchronous systems, Tech. Rep. #81-03-01, University of Washington, 1981.

[2] J. Cuny and L. Snyder, Conversion from data-flow to synchronous execution in loop programs, Tech. Rep. #CSD-TR-392, Purdue University, 1982.

[3] J. Cuny and L. Snyder, "Testing coordination for 'homogeneous' parallel algorithms", Proceedings of the 1982 International Conference on Parallel Processing, pp. 265-267, August, 1982.

[4] M. Garey and D. Johnson, *Computers and Intractability*, W. H. Freeman and Co., San Francisco, California, 1979.

[5] R. Karp and R.E. Miller, "Properties of a model for parallel computations: determinacy, termination, queuing", *SIAM J. Appl. Math.* 14, pp.1390-1411 November, 1966.

[6] H.T. Kung and C.E. Leiserson, Systolic arrays (for VLSI), Tech. Rep. CMU-CS-79-103, Carnegie-Mellon University, 1979.

[7] R. J. Lipton, R. E. Miller, and L. Snyder, "Synchronization and computing capabilities of linear asynchronous structures", *JCSS* 14, pp. 49-72, February, 1977.

[8] J. Peterson, "Petri nets", *Comp. Surveys* 9, pp.223-252, September, 1977.

[9] L. Snyder, Introduction to the Configurable, Highly Parallel Computer, *Computer* 15, pp. 65-82, January, 1982.

ERRATA

In the original version of this paper, an IC system was determined by a function A and a set of infinite sequences $V_1, V_2, \ldots, V_m$. Using that formulation, a PE has only one possible behavior and the system as a whole has only one possible execution sequence. We had intended a more versatile model.

In the corrected version, an IC system is determined by the function A and a set of finite state machines, $M_1, M_2, \ldots, M_m$, each describing the behavior of a single PE. It is a specific *execution sequence* that is determined by the set $V_1, V_2, \ldots, V_m$. To avoid explicit representation of process termination, we assume that each V_i is of the form $\alpha(\varphi)^\infty$ with α a prefix of some element of $L(M_i)$. Execution sequences are defined as before.

Finally, we intend for the operations $r_{i,\sigma}$ and $w_{j,\sigma'}$ to correspond only if $\sigma = \sigma'$. To enforce the matching, we consider just *legal computation sequences* in which for all i, j, and k

$$q^k_{i,j} \in \Sigma^* \cup (\Sigma^{-1})^*.$$

This restriction allows us to express the dependency of branching on transmitted values because, unless the corresponding reads and writes match, some link will have a status in $\Sigma^* \cup (\Sigma^{-1})^*$.

The corrected version of this paper will appear in *Algorithmically-Specialized Computers*, L. Snyder, L. Seigel, H. Seigel, and D. Gannon (eds.).