Yale University
Department of Computer Science

Design of Systolic Algorithms for Large Scale Multiprocessors

Jingke Li, Marina Chen, and Mark Young

YALEU/DCS/TR-513
Revised October 1988

Design of Systolic Algorithms for Large Scale Multiprocessors

Jingke Li[†] Marina C. Chen[†] Mark F. Young[‡]

July 1988

Abstract

One of the many challenges of a parallel-program compiler is to map an algorithm across a large number of processors while minimizing increased communication overhead. In this paper, a systematic method for generating systolic programs for large scale multiprocessors is described. The method applies to a subclass of programs called quasi-uniform recurrent equations, which is an extension to uniform recurrent equations appearing in literature. With a given optimality principle, our method will generate a (near) optimal mapping from the equations to a virtual machine with a given network topology. The solution on the virtual machine is then transformed to one for a concrete multiprocessor, based on the performance model of the target machine. A compiler implementing this method has been developed. It takes a set of quasi-uniform recurrent equations as its input and generates target systolic programs for the Intel iPSC/1 as its output. The performance model used by the compiler to fine-tune the target code as well as the performance results of the compiler generated code are discussed.

1 Introduction

Parallel processing can speed up computations substantially for many applications [3,4,5]. The appearance of the first generation of message-passing multiprocessors in recent years shows that it is not only desirable but also practical to build parallel machines. When talking about message-passing multiprocessors, we refer to machines such as the Cosmic Cube [21], the FPS T series [1], the Intel iPSC [2], the Connection Machine [11], and the WARP systolic processors [6].

One promising new architecture has already been proposed by Dally and Seitz [8,9]. The machine consists of a massive number of processing nodes interconnected into a mesh-like torus network. In contrast to present machines, this machine has a fine-grained architecture and it has a large diameter with low connectivity. Furthermore, the communication costs on this type of a machine is improving. As claimed in [8], the local node-to-node communication costs can "approach main memory access times of sequential computers." In order to achieve the greatest potential of efficiency provided by this type of multiprocessor, one can take advantage of fast local communication while avoiding the large diameter of the network.

[0†] Department of Computer Science, Yale University, New Haven, CT 06511.

[0‡] Floating Point Systems, Inc., P.O.Box 23489, Portland, OR 97223.

Large diameter, low connectivity multiprocessors have many properties in common with systolic arrays. There is a large class of important application algorithms, like numerical solutions to partial differential equations, matrix operations, signal processing, dynamic programming and some graph algorithms, which have the property that the data dependencies of the algorithms are very regular and are mostly local. These algorithms have been shown to be suitable for systolic processing. Many efficient systolic arrays for them have been designed. It is highly desirable to emigrate these algorithms to multiprocessors so that they can be executed efficiently not only on special purpose devices but also on more general and economical multiprocessors.

Low-level programming of multiprocessors, especially large-scale multiprocessors, can be tedious and error-prone. Ideally, programming is left to a high-level language compiler. A parallel-program compiler bears much more responsibility than a conventional compiler. One of the significant challenges for such a compiler is to map an algorithm across a large number of processors while minimizing communication overhead. Many methods for designing special purpose systolic arrays have been developed [7,10,14,16,17,19,22]. However, designing systolic programs for multiprocessors requires the target machine characteristics – communication latency between processors, ratio of processing speed versus communication speed, machine size, network topology, etc. – to be taken into account. In designing systolic arrays, the goal is to find good designs to be realized on VLSI chips. The size and the topology of the design to be constructed is tailored to heed the special-purposes of the algorithm in question.

In this paper, the problem of designing systolic algorithms for large-scale multiprocessors is explored. A systematic procedure for constructing linear mappings from an algorithm to a target network topology is described. The method is applicable to algorithms that are characterized by a set of recurrent equations called quasi-uniform recurrent equations (QURE) which are an extension to the well-known uniform recurrent equations (URE) [13] used in many previous works. Ideas similar to QURE have also been proposed by Rao [20]. Delosme and Ipsen have considered affine recursive equations which are a different extension to URE. The target machine is described by a performance model, in which computation cost, communication cost, granularity, and etc. are taken into account. The procedure for designing systolic programs consists of two parts. First, a (near) optimal linear mapping from a system of QURE to a virtual network is derived using linear programming plus a reasonable amount of heuristics to avoid the high cost of integer programming. The virtual network is defined based on the topology of the target machine. Optimality is measured against some given criteria. The program for the target machine is generated by choosing parameters that optimize the performance based on a model of the target machine.

The rest of this paper is organized as follows. For this article to be self-contained, section 2 presents preliminary notions of linear mappings and the necessary and sufficient conditions for the existence of a linear mapping. In Section 3, a procedure combining linear programming and heuristics for generating linear mappings is described. Section 4 discusses the relationship between the virtual machine and the target machine. In Section 5 we present a performance model for the Intel iPSC/1, a (coarse-grained) hypercube multiprocessor, and the discuss performance results of compiler generated target code. We conclude with a few remarks in Section 6.

2 Linear Mapping and the Mapping Equation

This section presents basic definitions and results concerning linear mappings. Readers can find alternative expositions in previous works [7,10,13,14,16,17,19].

2.1 Preliminaries:

2.1.1 Dependency Vectors:

Let $f(i_1, i_2, \ldots, i_n)$ be a recurrent equation defined on an n-dimension integral space Z^n:

$$f(i_1, i_2, \ldots, i_n) = \phi(f(j_1, j_2, \ldots, j_n), \ldots, f(j'_1, j'_2, \ldots, j'_n)). \tag{1}$$

Denote the domain of f by Ω. Each point in Ω represents a unit computation which is called a *computation point.* (Computation points are comparable to the names of processes or objects in other models of computation.) Based on the definition of f, a dependency relation among all the computation points can be defined.

Definition 2.1 *A computation point* $\mathbf{u}$ *is said to depend on a computation point* $\mathbf{v}$ *(denoted by* $\mathbf{u} \succ \mathbf{v}$*), if* $f(\mathbf{v})$ *appears on the right hand side of Equation 1, when* $f(\mathbf{u})$ *is on the left hand side. The difference of the two points,* $\mathbf{d} = \mathbf{u} - \mathbf{v}$*, is called a dependency vector (associated with* $\mathbf{u}$*).*

Denote the set of dependency vectors associated with a computation point $\mathbf{u}$ by $\mathcal{D}(\mathbf{u})$. For a general recurrent equation, $\mathcal{D}(\mathbf{u})$ may depend on $\mathbf{u}$. For example, in the following equation, at every odd-k point $\mathcal{D}(\mathbf{u}) = \{(1,1)\}$, while at every other point $\mathcal{D}(\mathbf{u}) = \{(-1,1)\}$:

$$f(i,k) = \begin{cases} f(i-1, k-1) + 1 & \text{if } k \text{ odd}, \\ f(i+1, k-1) + 1 & \text{otherwise.} \end{cases} \tag{2}$$

2.1.2 Quasi-Uniform Recurrent Equations:

There is a class of recurrent equations called *uniform recurrent equations* (URE) [13], in which $\mathcal{D}(\mathbf{u})$ is a finite constant set independent of $\mathbf{u}$ and the size of the domain. However, as far as linear mapping is concerned, URE is a little too restrictive in the sense that some non-UREs can still have linear mappings. In the following, we define QURE, a superset of URE, to make our result precise. The extension from URE to QURE is not substantial, most of the results developed for URE can be easily adapted to QURE.

Definition 2.2 *A recurrent equation* $f = \phi(f)$ *is quasi-uniform if the set of all dependency vectors,*

$$\mathcal{D} = \bigcup_{\mathbf{u}\, in\, \Omega} \mathcal{D}(\mathbf{u}),$$

is a finite constant set independent of the size of the domain.

The set of dependency vectors in D can be represented by an $n \times m$ matrix where n is the dimension of the QURE and m is the total number of dependency vectors in the set. The matrix constructed is called a *dependency matrix* of the QURE. Note, that m can either be greater or less than n. For example, a five-point discrete approximate of the Laplace equation in two dimension:

$$f(i,j,t{+}1) = (\ f(i{-}1,j,t) + f(i{+}1,j,t) + f(i,j{-}1,t) + f(i,j{+}1,t) - 4f(i,j,t)\ )/h^2 \qquad (3)$$

has five dependency vectors, which can be represented by a dependency matrix:

$$D = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

2.1.3 Communication Vectors:

We consider the class of parallel machines having regular multi-dimensional network structures. Processing elements are interconnected by communication channels or links. Typical network topologies include hexagonal networks and torus-connected k-ary n-cubes (hypercubes fall into this group). In such networks every node has the same set of communication links. For example, in a $2D$ square mesh, every node will have four links each connecting to its nearest neighbors in north, south, east and west directions, respectively.

Communication links are represented by *communication vectors.* If the dimension of the network is k, then each communication vector is of length $k{+}1$. The first k component corresponds to k spatial dimensions and the last component represents time. We assume that passing a message through a communication link takes one unit of time. Hence, the last component of every communication vector is 1. Furthermore, we represent time delays explicitly. A unit delay is represented by a communication vector $(0,\ldots,0,1)$. A *topology matrix* can be defined for each network such that each communication vector is a column vector. For convenience, we assume that the delay vector $(0,\ldots,0,1)$ is always the last column vector in a topology matrix.

2.2 Linear Mapping:

Let Ω denote the domain of an n dimensional QURE. Let $\mathcal{F}$ denote a linear mapping function from Z^n to Z^n, and let $\mathcal{F}_t$ denote the time component of $\mathcal{F}$. The function $\mathcal{F}$ maps each point in Ω to a point in the range $\mathcal{R}$ which represents a process to be executed on a certain node in the network. Linear mappings preserve the dependency relation among the computation points. We call the set of points in the range $\mathcal{R}$ with dependency relation "$\succ$" a *design* of the QURE. Any meaningful mapping function $\mathcal{F}$ has to satisfy the causality constraint in the time domain:

Definition 2.3 *A linear mapping satisfying the following two conditions is called a basic linear mapping.*

A1. *For all x, y in Ω, if $x \neq y$, then $\mathcal{F}(x) \neq \mathcal{F}(y)$;*
A2. *For all x, y in Ω, if y depends on x, then $\mathcal{F}_t(x) < \mathcal{F}_t(y)$.*

For a QURE, any computation point in its domain can be expressed by a linear combination of dependency vectors together with initial boundary points. The result of applying a linear mapping to a computation point, then, is equivalent to the result of applying the mapping to each corresponding dependency vector first and then taking the linear combination of them together with the mapping of the boundary points. We can concentrate on the mapping of dependency vectors rather than each individual computation point. For a QURE $f = \phi(f)$, if there exists a linear mapping for every dependency vector in the set $\mathcal{D}$, then the image of any computation point can be found accordingly since the set of dependency vectors associated with each individual computation point is a subset of $\mathcal{D}$.

A linear mapping from Z^n to Z^n can be represented by an $n \times n$ square matrix. Let T be such a matrix. T may contain elements that are rational. Let D be the dependency matrix of the QURE and C be the topology matrix of the network.

Definition 2.4 *Given a QURE with a dependency matrix D and a network with topology C, a square matrix T satisfying the following three conditions is called a valid linear mapping of D with respect to C.*

B1. *T is non-singular;*
B2. *For every column vector $\mathbf{d}$ in D, $\mathbf{t} \cdot \mathbf{d} > 0$, where $\mathbf{t}$ is the last row vector of T, which represents time ($\cdot$ is the inner product of vectors).*
B3. *(Validity w.r.t. topology C) For each column vector $\mathbf{d}$ in D, the resulting space-time vector $T\mathbf{d}$ must be a composition of communication vectors given by matrix C, where a composition is a linear combination with non-negative integral coefficients.*

2.3 The Mapping Equation:

A valid linear mapping defines a valid design with respect to a given network. The following proposition tells us that we can find all the valid designs by solving a matrix equation, and conversely, any valid mapping must satisfy this matrix equation.

Proposition 1 *Given a QURE with dependency matrix D $(n \times m)$ and a network topology matrix C $(n \times k)$, then an $n \times n$ matrix T is a valid linear mapping of D with respect to C if and only if T is non-singular and there exists a $k \times m$ non-negative integral matrix P satisfying*

$$TD = CP \tag{4}$$

and each column of P contains at least one positive element.

Proof:

($\Rightarrow$) Let T be a valid linear mapping of D with respect to C. Denote the last rows of T and C by $\mathbf{t}$ and $\mathbf{c}$, respectively. By Condition *B3*, for each column vector of D, $\mathbf{d}_j$, the vector $T\mathbf{d}_j$ is a composition of communication vectors. Therefore, it can be expressed as $C\mathbf{p}_j$ for some non-negative integral vector $\mathbf{p}_j$. Furthermore, since $\mathbf{t} \cdot \mathbf{d}_j = \mathbf{c} \cdot \mathbf{p}_j > 0$ (by Condition *B2*), $\mathbf{p}_j$ contains at least one positive element. Let P be a matrix containing all the vectors $\mathbf{p}_j$ as columns. If $P = (\mathbf{p}_1, \ldots, \mathbf{p}_m)$, then P satisfies Equation 4 and all it's conditions.

($\Leftarrow$) Since T is a non-singular matrix by our assumption, we only need to show that T satisfies Conditions *B2* and *B3*. Let P be a matrix satisfying the above equation and conditions. By definition of communication vector, $\mathbf{c}$ is a vector of all 1s. Since each column of P contains at least one positive element, we know that every element of vector $\mathbf{c}P$ is an integer greater than 0. This implies that T satisfies Condition *B2*. Furthermore, since P is non-negative and integral, for every column j of P, vector $C\mathbf{p}_j$ is a composition of communication vectors. Therefore, T satisfies Condition *B3*. Hence, T is a valid linear mapping of D with respect to C. *Q.E.D.*

Equation $TD = CP$ not only describes the mapping problem, but also provides some useful information about the resulting designs.

Routing information: Suppose that T and P are a solution to the equation. A valid linear mapping maps each dependency vector to a composition of communication vectors. The information about the composition is completely represented by the corresponding column of matrix P. For example, if there is a column $\mathbf{p}_j = (1, 2, 0, 1)$, then we know that dependency vector $\mathbf{d}_j$ is mapped to a path in the network which uses the first communication link once; the second twice and delay one time unit (recall that the last communication vector represents delays).

Timing information: We describe delay by an explicit communication vector. The advantage of doing this is that the timing for message passing is totally explicit. Therefore, we can derive completely self-synchronized designs in which the data needed for a computation will always get to the right place at the right time. Using explicit timing also makes it easier to define and search for time optimal designs.

2.4 Existence Result:

Definition 2.5 *A time direction for a QURE is an integral*[1] *vector* $\mathbf{t}$ *such that for all the dependency vectors* $\mathbf{d}$, $\mathbf{t} \cdot \mathbf{d} > 0$. *A QURE is consistent if it has a time direction.*

Intuitively, a time direction of a QURE is a common direction of all the dependency vectors since geometrically, condition $\mathbf{t} \cdot \mathbf{d} > 0$ means that the angle between vectors $\mathbf{t}$ and $\mathbf{d}$ is $< 90°$. The time direction is very close to the algebraic concept *polar vector* defined on a set of vectors with the exception that a polar vector does not require the dot product to be strictly greater than 0.

Not all QUREs have a time direction. However, whether a QURE is consistent can be determined without too much difficulty. The time complexity for solving this problem with a fixed n is polynomial in the total number of dependency vectors.

In a special case where the given QURE represents an iterative algorithm, the set of dependency vectors has a built-in time direction since the explicit iteration index, or time index, is monotonic with respect to the dependency relation. Hence, the vector corresponding to that index is a valid time direction. Take Equation 3 as an example. The third index of that equation, t, is an explicit

[1]This is not essential, we can use rational instead.

time index. Every computation point in the domain of that equation only depends on points which have smaller t values. Hence, the third components of all the dependency vectors are positive integers (actually, they are all 1 in this particular example). The vector $(0,0,1)$, which corresponds to the third index, is a time direction for that equation.

Definition 2.6 *The complementary vector of a communication vector* $\mathbf{c} = (c_1, \ldots, c_{n-1}, 1)$ *is the vector* $(-c_1, \ldots, -c_{n-1}, 1)$. *A network (represented by C) is symmetric if for any communication vector in C, its complementary vector is also in C.*

Intuitively, a network is symmetric if every physical link is bidirectional. Most existing parallel machines use symmetric networks. Some of them have two separate physical links; some use time multiplexing to realize bidirectional communications.

Theorem 1 *Given a QURE with dependency matrix D and any symmetric network topology C, a valid linear mapping of D with respect to C exists if the QURE is consistent.*

Corollary: *If a QURE has an explicit time index, then it has valid designs with respect to any symmetric network.*

The above theorem shows that as long as a QURE is consistent, it can be mapped to a symmetric network by a linear mapping, and iterative algorithms represented by a QURE can always be mapped to any symmetric network by a linear mapping.

3 Procedure for solving the mapping equation

Finding one solution to the mapping equation may not be very hard, but finding an optimal one can be very difficult. It depends on the complexity of the optimality criteria. If optimality criteria can be formulated in a linear form, then the problem of finding an optimal solution can be formulated as an integer programming problem. Otherwise, no method except exhaustively searching through the solution space has been developed [14].

In this section, we first discuss and compare several optimality criteria. Then, we describe a systematic method for finding an optimal solution to the mapping equation with a linear optimality criteria. Our method is based on linear programming and a reasonable amount of heuristics so as to avoid an expensive integer programming procedure. The trade-off of using a fast procedure instead of getting an optimal solution is based on the following two observations:

1. Experimentally, the heuristics have been able to find the optimal solution most of the time;
2. Near-optimal solutions at this level can be just as good due to processing stages to be performed later. Partitioning a large number of logical elements to a fixed size physical processor usually makes the advantage of optimal solution over near-optimal solution negligible.

3.1 Different Optimality Criteria:

There are many different optimality criteria for measuring systolic designs. The following are three examples (See [13,19] for the first two definitions):

OC1. Wavefront stages: Once a QURE is given, together with its index domain, a wavefront number can be computed for each node in its dependency graph which tells when it should be computed. The maximum wavefront number can be considered as an optimality criterion, i.e. a design in which every node is scheduled at a time slot which corresponds to its wavefront number can be considered as an optimal design. As a matter of fact, such a design can be derived from the dependency graph.

OC2. Minimal time steps w.r.t. linear mappings: Since a scheduling function achieving **OC1** may have a complicated form and may be costly to use, more practical optimality criteria can often be defined by putting constraints on scheduling functions to make sure they are economical to use. Criteria **OC2** is such that the scheduling function is restricted to be linear. A design is optimal according to **OC2** if it has the minimal number of time steps with respect to a linear scheduling function.

OC3. Shortest execution time w.r.t. a given target machine: The above two optimality criteria are defined independently of the target network topology. However, a design so defined may not correspond to one which runs the fastest on a target machine. For example, if we assume that a message traversing through a physical link takes a unit of time, and that an **OC2** optimal scheduling function is described by vector $(1,1)$, then if a target machine does not have diagonal channel (i.e. $(1,1)$ is not a communication vector), the corresponding optimal design may not be the one that runs the fastest. With a proper definition for the computation time of a machine, an optimal design can be defined to be the one which has the shortest execution time on the target network.

All of these three criteria can be expressed as objective functions. Suppose a pair T and P is a solution to the mapping equation (Equation (4)), and $\mathbf{t}$ is the time component of T. Then, the total execution time of the corresponding design is given by,

$$\text{exe\_time}(T,P) = \max_{\mathbf{x}\,\text{in}\,\Omega} (\mathbf{t}\cdot\mathbf{x}) - \min_{\mathbf{x}\,\text{in}\,\Omega} (\mathbf{t}\cdot\mathbf{x}). \tag{5}$$

Without losing generality, we can rename the origin in the problem index domain, so that Equation (5) can be simplified to

$$\text{exe\_time}(T,P) = \max_{\mathbf{x}\,\text{in}\,\Omega} (\mathbf{t}\cdot\mathbf{x}). \tag{6}$$

To obtain an optimal design, exe_time(T,P) must be minimized over all possible solutions of T and P. Solving this problem involves finding a solution for a system consisting of a large number of equations (comprising all the unknowns in T and P) and an unconstrained objective function. In this case, the optimality criteria can be approximated by a linear function, as discussed below.

3.2 The Searching Procedure:

The procedure for finding an optimal solution to the mapping equation consists of three steps. The first step involves finding an optimal time direction. Second, a mapping matrix T is constructed, and third a matching path matrix P is found. The input to the procedure is a set of dependency vectors of a consistent QURE and a set of communication vectors of a symmetric network, represented by matrices D and C, respectively. The following problem serves as an example:

Problem: Map a nine point discrete approximation of the Laplace equation to a symmetric hexagonal network.

$$\begin{aligned} f(i,j,t+1) = \; & (4f(i+1,j+1,t) + 4f(i+1,j-1,t) + f(i+1,j,t) + \\ & 4f(i-1,j+1,t) + 4f(i-1,j-1,t) + f(i-1,j,t) + \\ & 4f(i,j+1,t) + f(i,j-1,t) - 20f(i,j,t))/h^2. \end{aligned} \qquad (7)$$

The corresponding dependency matrix and topology matrix are:

$$D = \begin{pmatrix} -1 & -1 & -1 & 1 & 1 & 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & -1 & 1 & 0 & -1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix},$$

$$C = \begin{pmatrix} -1 & -1 & 1 & 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & -1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

Step 1. Finding an optimal time vector:

The first step is to find an integral vector $\mathbf{t} = (t_1, t_2, \ldots, t_n)$ such that $\mathbf{t} \cdot \mathbf{d}_i > 0$ for $D = (\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_m)$, and the corresponding schedule function produces minimal time steps.

To formulate the optimality criteria as linear objective functions, we use the following observation:

Observation: With respect to **OC2**, a time vector with a small norm is better than one with a large norm. (Define norm to be a vector $\mathbf{t} = (t_1, t_2)$ to be $|t_1| + |t_2|$.)

To see the above, we assume that the problem domain is a rectangular domain, with four corner points, $(0,0)$, $(m,0)$, $(0,n)$, and (m,n). Let us suppose that $\mathbf{t} = (a,b)$ is a time vector. Then, the timing step number for a point (x,y) is given by $ax + by$. The maximum time step number will be $|a|m + |b|n$. Therefore, the smaller the norm, the fewer the time steps. If the assumption of the rectangular domain does not hold, the ranges of the indices must be known and all the boundary points of the domain are tested to obtain the maximum time step number.

According to the above observation, the optimality criteria can be formulated as

$$\textit{minimizing } |t_1| + |t_2| + \cdots + |t_n|.$$

The problem of finding an optimal time vector can be formulated as a minimization problem,

$$D^T \begin{pmatrix} t_1 \\ t_2 \\ \cdots \\ t_n \end{pmatrix} \geq \begin{pmatrix} 1 \\ 1 \\ \cdots \\ 1 \end{pmatrix}, \tag{8}$$

Objective function: $f = |t_1| + |t_2| + \cdots + |t_n|$.

Note that function f is not linear and variables $t_1, \ldots, t_n$ are unbounded (they can be either positive or negative). Linear programming techniques cannot be applied directly to this system. Thus, the above system is transformed into the following equivalent system, which is in a standard form:

$$[D|(-D)]^T \begin{pmatrix} x_1 \\ \cdots \\ x_n \\ x'_1 \\ \cdots \\ x'_n \end{pmatrix} \geq \begin{pmatrix} 1 \\ 1 \\ \cdots \\ 1 \end{pmatrix}, \quad x_i \geq 0,\ x'_i \geq 0,\ i = 1, \ldots, n, \tag{9}$$

Objective function: $f = x_1 + \cdots + x_n + x'_1 + \cdots + x'_n$.

A solution of this new system can be transformed to a solution of the original system by letting $t_i = x_i - x'_i$ for $i = 1, \ldots, n$.

The system we posted above is actually an integer programming system. There is no known deterministic polynomial-time algorithm for solving it. Nevertheless, there are many approximation algorithms which use heuristics and run in polynomial time. We have implemented one algorithm which uses a combination of linear programming and heuristics to find an integral solution.

See Table 5 for a script of applying this step to Equation (7).

Step 2. Finding a mapping matrix T:

After obtaining **t**, a complete mapping matrix T can be constructed. The solution to this is not unique, adding any additional $n-1$ linearly independent vectors to **t** forms a non-singular T. The designs corresponding to different T's will have different spatial properties.

Qualified T's can be generated systematically one by one up to an *a priori* bound, which provides the user a chance of comparing different designs. Alternatively, optimization principles for space can be provided by the user, so optimal solutions can be found. One such principle is to minimize the determinant of T, which corresponds to minimizing the image domain size. Under this principle, the first T to try is the one formed by taking **t** and $n-1$ other vectors from $(1,0,\ldots,0), (0,1,\ldots,0), \ldots, (0,0,\ldots,1)$.

Table 6 shows the corresponding scripts and results of this step on the mapping of the Laplace equation.

Step 3. Finding an optimal P for the chosen T:

For each column of the unknown matrix P, form a linear system:

$$C\mathbf{p}_j \leq \begin{pmatrix} 1 & & & \\ & \cdots & & \\ & & 1 & \\ & & & k \end{pmatrix} T\mathbf{d}_j, \qquad j = 1, 2, \ldots, m, \tag{10}$$

$$\text{Objective function: } f = P_{1,j} + P_{2,j} + \cdots + P_{m-1,j}.$$

The parameter k (a positive integer) is called a *time dilation bound* which determines the maximum length of the image communication path of a dependency vector. For the example we chose, setting k to 4 or 5 is adequate. The optimality measure used here is the length of the image communication path of each dependency vector. Recall for a moment that the last column vector of matrix C represents time delay, therefore, the last component of each column of P, $P_{m,j}$, indicates the usage of local memory or message buffers to implement time delay. In the above cost function however, $P_{m,j}$ is not included. That means only the total number of communication links is to be minimized for each dependency vector. We certainly also want to minimize time delays, however, we will handle that issue in next step.

Table 7 shows a script of solving the first column of P. The complete result of this step is shown in Table 8.

Step 4. Remove excess time dilation:

The matrix P constructed in the previous step may still contain unnecessary time delay due to over estimating the time dilation parameter k. To further improve P, excess time dilation needs to be removed. This is done by reducing the time dilation parameter k on the left hand side of the equation and factoring out the constant vector accordingly from the last row of P on the right hand side, while keeping both sides of the equation to be equal and keeping P to be non-negative and integral.

For example, to optimize the P shown in Table 8, we first reduce the time dilation parameter k to 3. Accordingly, we factor out a constant vector $(1, 1, 1, 1, 1, 1, 1, 1, 1)$ from the last row of P. (In general, the constant vector may contain different integers). The result is shown in Table 9. We further reduce k to 2 and get a new P shown in Table 10. However, we cannot go any further since that will make P contain negative elements.

Remarks:

The above procedure finds a pair of T and P in time polynomial in the total number of dependency vectors and communication vectors. For practical applications, the dimension parameter n is usually a small constant (1, 2, or 3) which makes the procedure very inexpensive in practice.

4 Relation Between a Virtual and a Target Machine

The linear mapping discussed in previous sections requires that both the domain and the range are of the same dimensionality. In this section, we discuss the problem of matching the dimensionality and the size of a QURE and a multiprocessor. First, we construct a virtual network whose dimensionality and size match the dimensionality and size of the QURE . Then, after a design is generated by the mapping procedure, the virtual network solution is mapped to a target machine. Relative work can be found in [18,12,20,22].

4.1 Construction of a Virtual Network:

For our purpose, a virtual network must satisfy the following two conditions:

1. The virtual network preserves the topology of the target network so that any algorithm designed for the virtual network can be realized on the target machine by a simple mapping.
2. A virtual network is considered to be a general network based on the target network in the sense that any realization on the target network has a realization on the virtual network.

We show the construction of a virtual network through the following example where we will solve a 4-dimensional QURE on a 2-dimensional mesh. To match the dimensionality of the QURE, a space-time domain of dimensionality four is needed. This means a 3-dimensional virtual network needs to be augmented from the mesh.

1. Extend the mesh to infinity along each of the two dimensions.
2. Create a third dimension orthogonal to the previous two. Replicate the (infinite) mesh along the third dimension.
3. Set up neighboring communication links between the layers along the third dimension in such a way that the projection of these links on the first two dimensions results in the original square mesh. All such links are added.

The first two dimensions of the network preserve the structure of the target network. No new communication links are created in these dimensions. This guarantees that any algorithm designed for the virtual network can be realized on the target machine by simple projections. The augmented dimension is collapsed down into the time dimension of a single processor in the target machine.

Using communication vectors, the physical and the virtual network with respect to the example can be represented by the following matrices:

$$C_{old} = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix},$$

$$C_{new} = \begin{pmatrix} 1 & 1 & 1 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & -1 & -1 & -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 1 & -1 & 0 & 1 & -1 & 0 & 1 & -1 & 0 & 1 & -1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

4.2 Mapping a Virtual Network Solution:

For simplicity, assume that both the virtual network and the physical network are rectangular. Networks of other regular shapes can be handled similarly. Let the dimensionalities of the virtual network and the physical network be m and n ($m \geq n$), respectively. Represent the size of the virtual network by a vector $(s_1, s_2, \ldots, s_m)$, in which component s_i represents the size along the ith dimension. Similarly, let vector $(k_1, k_2, \ldots, k_n)$ represent the size of the physical network. Without loss of generality, we further assume that dimensions $n, n+1, \ldots, m-1$ of the virtual network are the augmented spatial dimensions (dimension m corresponds to time).

One way of mapping a virtual network to a physical network is to partition the computations with respect to a fixed time slice.

Denote the computation performed at time t at node $(x_1, \ldots, x_{m-1})$ in the virtual network by $f(x_1, \ldots, x_{m-1}, t)$. Then the set of computations corresponding to a fixed time slice t in the whole network can be expressed as follows:

$$\mathcal{F}_t = \{f(\ldots, x_i, \ldots, t) \mid x_i = 1, \ldots, s_i\}.$$

There are no dependency relations among these computations. Therefore, they can be executed in any sequential order of interleaving.

Reducing the dimensionality of a virtual network from m to n is done by projecting the augmented spatial dimensions to the time dimension. More specifically, the indices of the augmented dimensions are mapped to loop indices within a single processor. The order of executions of the computations in the loops are not essential. One simple construction of the loop is the following:

```
Code for node $(x_1, \ldots, x_{n-1})$ at time $t$:
  FOR $i_1 := 1$ TO $s_n$ DO
    FOR $i_2 := 1$ TO $s_{n+1}$ DO
      ...
        FOR $i_{m-n} := 1$ TO $s_{m-1}$ DO
          compute $f(x_1, \ldots, x_{n-1}, t)$.
```

After the dimensionality of the virtual network is reduced, the size of the virtual network can be compressed (if needed). We introduce a a function $\mathcal{G}$ to describe the compression. The input to this function is the address of a node in the virtual network and the output is the address of a node in the physical network. Function $\mathcal{G}$ indicates which physical processor of a virtual node is mapped to. Two methods for doing this are discussed below.

Grouping: It maps neighboring nodes in the virtual network into a physical node. It can be described by the following formula (recall that $(k_1, k_2, \ldots, k_n)$ which represents the size of the physical network):

$$\mathcal{G}(x_1, \ldots, x_n) = (\lfloor \frac{x_1 k_1}{s_1} \rfloor, \ldots, \lfloor \frac{x_n k_n}{s_n} \rfloor), \tag{11}$$

where $(x_1, \ldots, x_n)$ is a node in the virtual network.

Wrap-around: It effectively makes the physical network appear as an infinite *torus*. The corresponding formula is:

$$\mathcal{G}(x_1, \ldots, x_n) = (x_1 \bmod k_1, \ldots, x_n \bmod k_n). \tag{12}$$

After local grouping or wrap-around, each physical processor will represent a set of virtual nodes. A simple loop is needed to simulate these nodes. Again, the order of these simulations is not important since the corresponding computations are independent of each other with respect to the dependency relation. A possible scheduling loop is:

```
Code for node $(x_1, \ldots, x_{n-1})$ at time $t$:

   FOR each $(y_1, \ldots, y_{n-1})$ in set

         $\{(y_1, \ldots, y_{n-1}) \mid \mathcal{G}(y_1, \ldots, y_{n-1}) = (x_1, \ldots, x_{n-1})\}$

      compute $f(y_1, \ldots, y_{n-1}, t)$.
```

A nice property of the above compression methods is that the interprocessor communication patterns are still the same as they were before the compression, except that the volumes of the messages are increased.

4.3 Granularity versus Latency:

On a coarse-grained machine like the iPSC/1, a startup overhead is attached to each message transmission so it is more economical to pack small messages together to send them as one bigger packet, rather than sending them separately. Hence, another way of mapping a large design on a virtual network to a physical machine is to enlarge the unit data packet size. Instead of applying the linear mapping directly to the index points, we first decompose the index domain to form a collection of subdomains. Then, we apply the linear mapping to the space of subdomains. In other words, we consider a subdomain as a unit, and analyze dependencies between these subdomains. However, enlarging each subdomain may increase the initial waiting time (i.e. the latency) of a processor because it cannot start its computation unless it gets the result of the preceding node. The tradeoff between granularity and latency can be adjusted by the compiler based on the performance model introduced in next section.

5 A Performance Model

The mapping procedure presented above has been incorporated in a compiler for a high level parallel language Crystal. A user program written in Crystal is first transformed to a set of QUREs [7]. Then a set of dependency vectors are extracted from the QUREs to form a dependency matrix. A matrix representing the target network topology is also formed. The mapping procedure is applied to the dependency matrix and topology matrix to produce (near) optimal mappings. Finally, a target code is generated based on the mapping and the parameters obtained from the performance model of the target machine.

The performance model consists of characterizations of both the program and the target machine. Characterization of the program is obtained from the source program and the mapping described in the two previous sections. Once a model is established, it is then used to predict the performance of the program on the target machine with respect to a chosen optimization criterion.

5.1 Machine Profile

- p – the total number of processors.
- τ_p – the time it takes to perform a unit computation on a unit data element. A unit computation may mean an integer operation, a floating point operation, or a mixed sequence of both. A unit data element is either an integer or a floating point number. The exact value τ_p depends on the instruction cycle and the basic cycle time. In this paper, all timing is given in units of micro-seconds.
- τ_s – the startup time for transmitting a message.
- τ_c – the time it takes to transmit a unit data element to an adjacent processor.
- τ_{bc} – the time it takes to broadcast a unit data element to other processors. For hypercube multiprocessors, $\tau_{bc} = \tau_c \log(p)$.

5.2 Data Size

One of the factors that determines the amount of communications between processors is the amount of data assigned to each processor. The way data is partitioned is expressed by the mapping from virtual network to physical network. The parameters of the mapping are determined by optimizing the performance with the given performance model. The following is a list of parameters relating to the data size.

- N – total number of data elements in the program. For example, in Program Matrix Multiplication, N equals the total number of matrix elements.
- n – total number of data elements on a given processor. If the data is evenly partitioned over all of the p processors, $n = N/p$.
- n_{nc} – total number of data elements to be transmitted from a given processor to its adjacent processor (neighboring communications). Note, that n_{nc} is a function of n.
- n_{bc} – total number of data elements to be broadcasted from a given processor to other processors. Similarly, n_{bc} is also a function of n.

- k_{nc} – total number of messages for neighboring communications.
- k_{bc} – total number of messages for broadcasting.

5.3 Performance of Individual Processors

- $t_p(i)$ – the computation time. It is a function of n:
$$t_p(i) = \tau_p \times n.$$
- $t_{nc}(i)$ – the time spent on neighboring communications,
$$t_{nc}(i) = \tau_s \times k_{nc} + \tau_c \times n_{nc}.$$
- $t_{bc}(i)$ – the time spent on broadcasting,
$$t_{bc}(i) = \tau_s \times \log(p) \times k_{bc} + \tau_c \times \log(p) \times n_{bc}.$$
- $t_l(i)$ – the starting latency time. In our model, we assume that all processors start their clocks at the same time. However, if one processor needs data from other processors in order to execute, it cannot start until such data arrive. The parameter t_l is defined to be the time interval from when the clock is started until the execution begins. The interval t_l certainly varies over processors. For a pipelined algorithm, for instance, the latency, t_l depends on the stage of a processor in the pipe. For processor i, we denote its stage by $\kappa(i)$ and the delay at each stage by Δ. Then,
$$t_l(i) = \kappa(i) \times \Delta.$$
- $t(i)$ – the total elapsed time,
$$t(i) = t_p(i) + t_{nc}(i) + t_{bc}(i) + t_l(i).$$

5.4 Global Performance Parameters

- T_s – the sequential elapsed time. It is defined as the total elapsed time executing the program on one single processor, for example Processor 1,
$$T_s = t(1) = t_p(1) \quad \text{since } t_c(1) = t_{bc}(1) = 0.$$
- T_e – the parallel elapsed time. It is defined as
$$T_e = \max_i t(i).$$
- T_p – the average computation time,
$$T_p = \frac{1}{p}\sum_i t_p(i).$$

- T_c – the average communication time,

$$T_c = \frac{1}{p}\sum_i (t_{nc}(i) + t_{bc}(i)).$$

- r – speedup,

$$r = T_s/T_e.$$

5.5 Target Program

The target machine of our first version of the Crystal compiler is an Intel iPSC/1 hypercube (with 32 nodes). The coarse-grained nature of this particular machine prevents us from taking full advantages of the systematic mapping procedure described above. For instance, the routing matrix P which tells how a message is to be routed will be of no use on the iPSC/1, since message routing on this machine is done at a very low level, over which our compiler does not have control. What is interesting is the trade-off between communication and latency. We illustrate this tradeoff by a matrix-matrix multiplication (MM) program compiled from the following source:

$$\begin{aligned}
c(i,j,k) \text{ over } N &= \ll k = 0 \rightarrow 0, \\
&\quad (n \geq k) \text{ and } (k > 0) \rightarrow c(i,j,k-1) + a(i,j,k) * b(i,j,k) \gg, && (13) \\
a(i,j,k) \text{ over } N &= \ll j = 0 \rightarrow a0(i,k), \\
&\quad (n \geq j) \text{ and } (j > 0) \rightarrow a(i,j-1,k) \gg, && (14) \\
b(i,j,k) \text{ over } N &= \ll i = 0 \rightarrow b0(k,j), \\
&\quad (n \geq i) \text{ and } (i > 0) \rightarrow b(i-1,j,k) \gg, && (15) \\
N &= M \times M \times M, \quad M = 1, \ldots, n.
\end{aligned}$$

For this example, the problem domain specified by index domain M, is partitioned first, then a linear mapping is applied. The partitioning is done by slicing the domain along all those dimensions to form a collection of almost equal-sized submatrices. Due to the uniformity of dependency in QURE, the dependencies between data associated with these submatrices are represented by the same set of dependency vectors which represent the dependencies between array elements. The granularity of the unit data element, i.e. the size of the submatrices, is controlled by a parameter. Based on the performance model introduced above, the Crystal compiler can then find a value for the parameter such that the total execution time is optimized. In the following, we will show that the optimal value produced by the compiler in fact agrees with the optimal value obtained from the experiment.

5.6 Performance Model of the Program

Suppose the input matrices are of sizes $M \times M$. According to the program (Equations (13)–(15)), the computation consists of M time steps. The index domain of this algorithm is a 3-dimensional cube, which is of size $M \times M \times M$. The p processors of the target machine are organized as a two-dimensional mesh of size $p_1 \times p_2$. The following parameters can be adjusted at compile time:

- k – the block size along the time axis. The total computation is divided into M/k iterations, each iteration consists of k time steps.
- p_1, p_2 – the number of processors along each dimension of the mesh ($p = p_1 \times p_2$).
- n_1, n_2 – the size of the sub matrix size. The index domain is thus decomposed into a collection of subdomains, each of size $n_1 \times n_2 \times k$. These subdomains are the unit data packets.

From these parameters, granularity is defined as the size of the unit data packet, i.e. $n_1 \times n_2 \times k$.

When $(M/n_1) > p_1$, a wrap-around of data element will occur along the first dimension of the mesh of the processors. Similarly, $(M/n_2) > p_2$, a wrap-around will occur along the second dimension. To simplify the analysis, we assume that the parameters are always chosen in a way that M/n_1 divides p_1 and M/n_2 divides p_2. The overall performance of the program can be obtained as follows. First, based on the size of the index domain, the total number of data elements is,

$$N = M^3.$$

The number of elements on each processor is therefore,

$$n = M^3/p.$$

The unit data element of the program is a floating point number. The unit computation involves several array references and two floating point operations (e.g. $c[i][j][k] = a[i][j][k] \times b[i][j][k] + c[i][j][k-1]$). On the Intel iPSC/1, a floating point operation costs $10\mu s$. So the time it takes to perform a unit computation on a unit data element is,

$$\tau_p = 30.$$

The message transmitting startup time on the iPSC/1 is $2ms$,

$$\tau_s = 2000.$$

The average cost of transmitting one floating point number (which is 4 bytes) is $8\mu s$. So,

$$\tau_c = 8.$$

Since a unit data packet size is $n_1 n_2 k$, there are $n/(n_1 n_2 k)$ unit data packets on each processor. Two message transmissions are associated with each unit data packet, one to the north neighbor and one to the east neighbor. The total number of messages for neighboring communication is

$$k_{nc} = 2n/(n_1 n_2 k) = 2M^3/(n_1 n_2 k p).$$

The size of these messages is either $n_1 k$ or $n_2 k$, because only the data elements in two boundary planes of a unit data packet need to be transmitted. Therefore, the total number of data elements to be transmitted is

$$n_{nc} = (n_1 k)n/(n_1 n_2 k) + (n_1 k)n/(n_1 n_2 k) = (n_1 + n_2)M^3/(n_1 n_2 p).$$

The time that a processor spends on neighboring communication is,

$$\begin{aligned} t_{nc}(i) &= \tau_s \times k_{nc} + \tau_c \times n_{nc} \\ &= 2000 \times 2M^3/(n_1 n_2 k p) + 8 \times (n_1 + n_2) M^3/(n_1 n_2 p) \\ &= 4000 M^3/(n_1 n_2 k p) + 8(n_1 + n_2) M^3/(n_1 n_2 p). \end{aligned}$$

The computation time is,

$$t_p(i) = 30M^3/p.$$

The unit latency equals the time a processor spends on computing a unit data packet,

$$\Delta = \tau_p(n_1 n_2 k).$$

The startup latency time is therefore,

$$t_l(i) = \kappa(i) \times (30 n_1 n_2 k).$$

Since there is no broadcasting, communication in this program is,

$$t_{bc}(i) = 0.$$

Therefore, the total elapsed time for an individual processor is,

$$t(i) = 30M^3/p + 4000M^3/(n_1 n_2 k p) + 8(n_1 + n_2)M^3/(n_1 n_2 p) + 30 n_1 n_2 k \kappa(i),$$

and the total elapsed time for the multiprocessor is,

$$\begin{aligned} T_e &= \max_i t(i) = \alpha_1 + \alpha_2 + \alpha_3 + \alpha_4, \text{ where,} \\ \alpha_1 &= \max_i t_p(i) = 30M^3/p, \\ \alpha_2 &= \max_i \tau_s k_{nc} = 4000M^3/(n_1 n_2 k p), \\ \alpha_3 &= \max_i \tau_c dsn_{nc} = 8(n_1 + n_2)M^3/(n_1 n_2 p), \\ \alpha_4 &= \max_i t_l(i) = 30 n_1 n_2 k \max(\kappa(i))) = 30 n_1 n_2 k (p_1 + p_2). \end{aligned} \qquad (16)$$

From this model, we examine the effect of various parameters on the performance.

5.7 Effects of Parameters on Performance

1. Changing p_1, p_2 – The only term in T_e involving p_1 and p_2 is $(p_1 + p_2)$, which minimizes when $p_1 = p_2 = \sqrt{p}$. This says that a square-mesh organization of processors is better than other types of organizations. See Table 1 for the experimental results.

2. Changing the granularity – The granularity of a unit data packet is controlled by three parameters n_1, n_2, and k. We first analyze the collective effect of these parameters, i.e. we would like to see how the product, $g = n_1 n_2 k$, as one term affects the performance. In Equation(16), α_1

does not affect, while α_3's effect is dominated by those of α_2 and α_4. Here, we want to see that in terms of minimizing $(\alpha_2 + \alpha_4)$, what the value of g should be. For this purpose, we form

$$f(g) = 4000M^3/gp + 60g\sqrt{p}.$$

To obtain the minimum value of the above function, we differentiate f with respect to g and solve equation

$$f'(g) = -4000M^3/(pg^2) + 60\sqrt{p} = 0,$$

which yields

$$g = \sqrt{\frac{4000M^3}{60p\sqrt{p}}}. \tag{17}$$

3. Changing n_1, n_2 – Now suppose we fix the product $n_1 n_2 k$ to the optimal value given by Equation(17). Under this constraint, larger n_1 and n_2 is better, since that will make α_3 smaller. The maximum values of n_1 and n_2 are M/p_1 and M/p_2, respectively. And by Point 1 discussed above, $p_1 = p_2 = \sqrt{p}$, $n_1 = n_2 = M/\sqrt{p}$.

4. Changing k – Effect of k on the performance is constrained by other parameters. If we let $p_1 = p_2 = \sqrt{p}$ and $n_1 = n_2 = M/\sqrt{p}$, then we can derive the optimal k from Equation(17),

$$k = \frac{g}{n_1 n_2} = (\frac{\sqrt{p}}{M})^2 \sqrt{\frac{4000M^3}{60p\sqrt{p}}} = \frac{10\sqrt{6}}{3} M^{-1/2} p^{1/4}. \tag{18}$$

What we found from the above analysis is that the simple block-partitioning (corresponding to letting $n_1 = n_2 = M/\sqrt{p}$) of the index domain is good enough for minimizing T_e. The more complicated wrap-around partitioning is not needed. This situation is due to the contribution of message startup time α_2.

Now let's take a concrete example to see how good the theoretically derived k is, comparing against the experimental one obtained from running the program on the real machine. Let $p = 4$ and $M = 50$. We have

$$k = \frac{10\sqrt{6}}{3} 50^{-1/2} 4^{1/4} \approx 1.6.$$

Take $k = 2$,

$$\begin{aligned} T_e &= 30 \cdot 50^3/4 + 4000 \cdot 50/2 + 16 \cdot 50^2/2 + 60 \cdot 2 \cdot 50^2/2 \\ &= 937500 + 100000 + 20000 + 150000 = 1197500 \ \ (\mu s). \end{aligned}$$

In order to compare the performance predicted by the model and the actual experimental result, we define the percentage of extra time taken. For each value of k, over the optimal time:

$$e(k) = \frac{T(k)}{T(\overline{k})} \times 100\%$$

where $\overline{k}$ denotes the optimal k value. Table 2 summarizes the result for $M = 50$ and $p = 4$.

The performance of the target code on the largest hypercube used in our experiments is summarized in Table 3.

(p_1, p_2)	(1,16)	(2,8)	(4,4)	(8,2)	(16,1)
T_e	1389	1185	970	1076	1306

Table 1: Effect of Changing p_1, p_2.

k	50	25	17	13	10	7	5	4	3	2	1
$e(k)$ (model)	3.92	2.36	1.86	1.61	1.43	1.25	1.13	1.08	1.03	1.00	1.02
$e(k)$ (exper)	1.76	1.33	1.19	1.12	1.07	1.03	1.01	1.00	1.00	1.01	1.10

Table 2: Comparison of the predicted performance and the actual performance.

NCUBE (p=128)				
program	T_p	T_c	T_e	r
Pipelined MM	14317	5075	21060	87.0
Block MM	721	171	980	94.2
Pipelined LUD	16240	5721	24220	85.8

Table 3: Performances of three programs on 128-node NCUBE. Matrix-size for pipelined MM and LUD is 200 $\times$ 200; for Block MM is 128 $\times$ 128.

6 Concluding Remark

In this paper we demonstrate techniques for synthesizing systolic algorithms. These techniques are applicable and they have proven to be useful in generating efficient target code for general purpose multiprocessors. In dealing with such machines, we take into account realistic constraints, such as processor and memory limitation and latency of communications between processors. For a parallel program to obtain good performance, we need to deal with the issue of load balancing among processors. Thus, optimization for code generation must be based on a realistic performance model of a particular problem on a particular machine.

We have shown such code generation by partitioning the index domain over which the quasi-uniform recurrent equations are defined, and we have chosen specific parameters based on a performance model. The performance model is established as follows: given measures of the execution time of the integer, floating-point, and other instructions, a compiler can examine the code and give an estimate of the total execution time. Similarly, given the unit communication time and the message startup time of a particular machine, a compiler can estimate for a given program the size of data transmitted between processors, and produce an estimate of the communication time.

The success of compiling the class of quasi-uniform recurrent equations to multiprocessor programs is encouraging and we are pursuing techniques for partitioning, task distribution, and performance modeling for a broader class of problems.

Acknowledgment

We would like to thank Floating Point Systems, Inc. for their generous support during the summer of 1986 where part of this work was conducted. Our thanks also to the many referees for their helpful comments, and to Eileen Connolly for her editorial efforts. We would also like to acknowledge John Johnston for valuable discussions in the past while collaborating with us in implementing the mapping procedure. The generous support by the Office of Naval Research under Contract No. N00014-86-K-0310 is gratefully acknowledged.

References

[1] *FPS T-series Brochures and Application Software Reference Material, Floating Point Systems, Inc.*, 1986.

[2] *iPSC Brochures and Application Software Reference Material, Intel Scientific Computers*, 1985.

[3] *Proceeding of the First Conference on Hypercube Concurrent Computers and Applications*, 1986.

[4] *Proceeding of the Second Conference on Hypercube Concurrent Computers and Applications*, 1987.

[5] *Proceeding of the Third Conference on Hypercube Concurrent Computers and Applications*, 1988.

[6] M. Annaratone, E. Arnould, T. Gross, H.T. Kung, M. Lam, O. Menzilcioglu, and J.A. Webb. The WARP computer: architecture, implementation, and performance. *IEEE Transaction on Computers*, C-36(12), December 1987.

[7] M. C. Chen. Synthesizing systolic designs. In *Proceedings of the Second International Symposium on VLSI Technology, Systems, and Applications*, pages 209–215, May 1985.

[8] W.J. Dally. *A VLSI Architecture for Concurrent Data Structures.* PhD thesis, Caltech, March 1986.

[9] W.J. Dally and C.L. Seitz. The torus routing chip. *Journal of Distributed Systems*, 1(3), 1986.

[10] J.-M. Delosme and I.C.F. Ipsen. An illustration of a methodology for the construction of efficient systolic architectures in vlsi. In *Proceedings of the 1985 International Symposium on VLSI Technology, Systems and Application*, May 1985.

[11] W.D. Hillis. *The Connection Machine.* MIT Press, 1985.

[12] J.J.Navarro, J.M.Llaberia, and M.Valero. Partitioning: an essential step in mapping algorithms into systolic array processors. *Computer*, 20(7), July 1987.

[13] R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *JACM*, 14(3):563–590, 1967.

[14] G. Li and B.W. Wah. The design of optimal systolic arrays. *IEEE Trans. on Computers*, C-34(1), Jan. 1985.

[15] D.G. Luenberger. *Introduction to Linear and Nonlinear Programming.* Addison-Wesley, 1973.

[16] W.L. Miranker and A. Winkler. Spacetime representations of systolic computational structures. *Computing*, 32, 1984.

[17] D.I. Moldovan. On the design of algorithms for vlsi systolic arrays. *Proceedings of the IEEE*, 71(1), Jan. 1983.

[18] D.I. Moldovan and J.A.B. Fortes. Partitioning and mapping algorithms into fixed-size systolic arrays. *IEEE Trans. Computers*, C-35(1), Jan. 1986.

[19] P. Quinton. Automatic synthesis of systolic array from uniform recurrence equations. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, June 1984.

[20] S.K. Rao. *Regular Iterative Algorithms and their Implementation on Processor Arrays.* PhD thesis, Stanford Univ., October 1985.

[21] C.L. Seitz. The cosmic cube. *CACM*, 28(1), Jan. 1985.

[22] Y. Wong and J-M. Delosme. Optimal systolic implementations of n-dimensional recurrences. In *IEEE Internatinal Conference on Computer Design: VLSI in Computers*, 1985.

Appendix 1:

Proof of Theorem 1:

($\Rightarrow$) Let T denote a valid linear mapping and let $\mathbf{t}$ denote the last row of T. By definition, for every dependency vector $\mathbf{d}$, $T_n \cdot \mathbf{d} > 0$. If T_n is an integral vector, it is a time direction itself. Otherwise, we form a vector $\mathbf{t}' = c\mathbf{t}$ where c is the least-common-multiple of the denominators of all rational components of $\mathbf{t}$. Then, $\mathbf{t}$' is integral and $\mathbf{t}' \cdot \mathbf{d} > 0$. Therefore, $\mathbf{t}$' is a time direction for the QURE. Hence the QURE is consistent.

($\Leftarrow$) Let $\mathbf{t}$ be a time direction. Since for every dependency vector $\mathbf{d}$, $\mathbf{t} \cdot \mathbf{d} > 0$, the angle between vector $\mathbf{t}$ and every $\mathbf{d}$ is $< 90°$. Define a hyperplane L which passes the origin and perpendicular to $\mathbf{t}$. Then, all the dependency vectors $\mathbf{d}$ lie on one and the same side of L. Find in L $n-1$ linearly independent vectors with rational components. (This can always be done, since L is of dimension $n-1$). Together with vector $\mathbf{t}$, these vectors form a basis for the whole dependency vector space. Every $\mathbf{d}$ can be expressed as a linear combination of these basis vectors with rational coefficients and the coefficient corresponding to $\mathbf{t}$ is positive. Furthermore, by properly scaling the norms of these basis vectors, we can express all the dependency vectors $\mathbf{d}$ with integral coefficients. Denote the basis vectors by square matrix D_S (let $\mathbf{t}$ be the last column). Then we have

$$D = D_S A$$

for some integral matrix A, whose last row consists of only positive integers.

On the other hand, we can find n linearly independent communication vectors from C. Denote them by C'. We have

$$C' = CB$$

for some 0-1 matrix B (B selects the corresponding columns to form C').

Now let T be a mapping matrix such that

$$TD_S = C'$$

then

$$TD = TD_S A = C'A = CBA = C(BA).$$

Matrix $Q = BA$ contains only integral elements, but the elements are not necessarily all non-negative. However, since C represents a symmetric network, we can define a new matrix P based on Q as follows:

$$P(i,j) = \begin{cases} Q(i,j) - Q(k,j) & \text{if } Q(i,j) > Q(k,j), \\ 0 & \text{otherwise.} \end{cases}$$

where the ith column vector and the kth column vector of C are complementary communication vectors of each other.

It's easy to verify that

$$TD = CP$$

and P satisfies all the conditions stated in the theorem. Hence T is a valid linear mapping of D with respect to C. *Q.E.D.*

Appendix 2:

The following scripts show a few snap shots of the process of solving the mapping equation by integer programming routines for the example given in Section 3. See [15] for details about linear programming method. The program implementing the procedure is written in T (a dialect of Lisp).

```
>(ppmtx d)
   -1  -1  -1   1   1   1   0   0   0
   -1   1   0  -1   1   0  -1   1   0
    1   1   1   1   1   1   1   1   1

>(ppmtx c)
   -1  -1   1   1   0   0   0
   -1   0   1   0  -1   1   0
    1   1   1   1   1   1   1
```

Table 4: Input matrices D and C.

```
> (mp d c)

Step 1. Finding a time direction:

First Tableau:
   -1  -1  -1   1   1   1   0   0   0   1   0   0   0   0   0   1
   -1   1   0  -1   1   0  -1   1   0   0   1   0   0   0   0   1
    1   1   1   1   1   1   1   1   1   0   0   1   0   0   0   1
    1   1   1  -1  -1  -1   0   0   0   0   0   0   1   0   0   1
    1  -1   0   1  -1   0   1  -1   0   0   0   0   0   1   0   1
   -1  -1  -1  -1  -1  -1  -1  -1  -1   0   0   0   0   0   1   1
   -1  -1  -1  -1  -1  -1  -1  -1  -1   0   0   0   0   0   0   0
X = (0 0 0 0 0 0 0 0 0 1 1 1 1 1 1),   S = (10 11 12 13 14 15)
f = 0

New Tableau:
    0   0   0   2   2   2   1   1   1   1   0   1   0   0   0   2
    0   2   1   0   2   1   0   2   1   0   1   1   0   0   0   2
    1   1   1   1   1   1   1   1   1   0   0   1   0   0   0   1
    0   0   0  -2  -2  -2  -1  -1  -1   0   0  -1   1   0   0   0
    0  -2  -1   0  -2  -1   0  -2  -1   0   0  -1   0   1   0   0
    0   0   0   0   0   0   0   0   0   0   0   1   0   0   1   2
    0   0   0   0   0   0   0   0   0   0   0   1   0   0   0   1
X = (1 0 0 0 0 0 0 0 0 2 2 0 0 0 2),   S = (10 11 1 13 14 15)
f = -1
Done.
Time vector found:  0    0    1
```

Table 5: Finding a time direction.

```
Step 2. Constructing T:
T:
    1    0    0
    0    1    0
    0    0    1
Is this T OK (y/n)? y

Input time dilation (a positive integer): 4
```

Table 6: Constructing T and setting time dilation bound.

```
Step 3. Finding P:

------------------------- For Column 1:

Slack Tableau:
    1    1   -1   -1    0    0    0    1    0    0    1
    1    0   -1    0    1   -1    0    0    1    0    1
    1    1    1    1    1    1    1    0    0    1    4
   -3   -2    1    0   -2    0   -1    0    0    0   -6
X = (0 0 0 0 0 0 0 1 1 4),   S = (8 9 10)
f = 6

New Tableau:
    1    1   -1   -1    0    0    0    1    0    0    1
    0    1    2    1    0    2    1    0   -1    1    3
    0   -1    0    1    1   -1    0   -1    1    0    0
    0    0    0    0    0    0    0    1    1    1    0
X = (1 0 0 0 0 0 3 0 0 0),   S = (1 7 5)
f = 0
Is this Tableau OK? y

Initial Tableau:
    1    1   -1   -1    0    0    0    1
    0    1    2    1    0    2    1    3
    0   -1    0    1    1   -1    0    0
    0    1    2    1    0    2    0   -1
X = (1 0 0 0 0 0 3),   S = (1 7 5)
f = 1
Done.
```

Table 7: Constructing 1st column of P.

```
P:
    1   0   0   0   0   0   0   0   0
    0   1   1   0   0   0   0   0   0
    0   0   0   0   1   0   0   0   0
    0   0   0   1   0   1   0   0   0
    0   0   0   1   0   0   1   0   0
    0   1   0   0   0   0   0   1   0
    3   2   3   2   3   3   3   3   4
```

Table 8: A matrix P resulting from Step 3.

```
Step 4. Removing excess time dilation:
T:
    1   0   0
    0   1   0
    0   0   3
P:
    1   0   0   0   0   0   0   0   0
    0   1   1   0   0   0   0   0   0
    0   0   0   0   1   0   0   0   0
    0   0   0   1   0   1   0   0   0
    0   0   0   1   0   0   1   0   0
    0   1   0   0   0   0   0   1   0
    2   1   2   2   2   2   2   2   3
```

Table 9: Removing excess time dilation.

```
Final result:
T:
    1   0   0
    0   1   0
    0   0   2
P:
    1   0   0   0   0   0   0   0   0
    0   1   1   0   0   0   0   0   0
    0   0   0   0   1   0   0   0   0
    0   0   0   1   0   1   0   0   0
    0   0   0   1   0   0   1   0   0
    0   1   0   0   0   0   0   1   0
    1   0   1   0   1   1   1   1   2
```

Table 10: Final result.