Yale University
Department of Computer Science

Formal Derivation of an Efficient Parallel Gauss-Seidel Method on a Mesh of Processors

J. Allan Yang Young-il Choo

YALEU/DCS/TR-870
October, 1991

This work has been supported in part by the Office of Naval Research under Contract No. N00014-91-J-1559 and National Science Foundation under Contract No. CCR-8807426.

Formal Derivation of an Efficient Parallel Gauss-Seidel Method on a Mesh of Processors

J. Allan Yang Young-il Choo

Department of Computer Science
Yale University
New Haven, CT 06520-2158
yang@cs.yale.edu, choo@cs.yale.edu

September, 1991

Abstract

This paper presents a formal derivation of a highly efficient parallel Gauss-Seidel method for machines based on a two-dimensional mesh of processors. It illustrates a framework which formalizes the process of mapping and scheduling a high level algorithm onto a particular target parallel architecture. We start from a simple and easy-to-verify initial program. Equational transformations are then applied (1) to partition the abstract problem onto processors, (2) to make communication among processors explicit, (3) to schedule processors at the earliest possible time step, and finally (4) to map virtual processor onto physical processors for perfect processor utilization. All the derivation steps are verified to preserve equalities so that the derived program is proved to be equivalent to the initial program. Using the same methodology, the result can be adopted naturally for hypercube and shared memory parallel machines.
Keywords: Program synthesis, formal program derivation, mapping and scheduling, equational program transformation, parallel programming paradigm.

1 Introduction

Gauss-Seidel method [5] is a fast numerical method for solving partial differential equations by iterative finite difference over a two-dimensional mesh of points:

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

where $\phi^{(k)}_{(i,j)}$ is the value at mesh point (i,j) at the k-th iteration. Initial mesh values (e.g., when $k = 0$) are given and values on boundary mesh points are fixed. The sequential implementation of Gauss-Seidel can be done by straightforward nested iteration. However, finding an efficient parallel implementation is much more difficult because of additional issues such as partitioning problem domain, coordinating interprocessor communication, scheduling processors, and improving processor utilization, etc.

This paper presents a formal derivation of a highly efficient parallel Gauss-Seidel method for machines based on a two-dimensional mesh of processors. It illustrates a framework which formalizes the process of *mapping* and *scheduling* a high level algorithm onto a particular target parallel architecture. We start from a simple high level functional specification of the Gauss-Seidel method as the initial program, which is verified easily. The initial program is defined over an abstract domain without any implementation details. A one-to-one mapping is then used to partition the abstract domain onto a mesh of processors. A new program defined over the explicit processors and their local partitions are derived by a new transformation technique called *reshape*. The derivation steps are verified to preserve equalities so that the derived program is equivalent to the initial one. We then make the interprocessor communication explicit by doing dependency analysis. Following that, wavefronts of the computation are used to schedule processors to start computing at the earliest possible time. Again, a one-to-one mapping is used to specify the scheduling and thus reshape is applied again to derive a new program with explicit scheduling. Finally, we map several virtual processors to one physical processor so that perfect processor utilization is achieved.

This paper is organized as follows. Section 2 briefly describe the language Crystal, which we use to express our programs. Section 3 gives the initial program. Section 4 gives the partition mapping and derives the new program with explicit processors and local partitions. Section 5 analyzes data dependencies and does case analysis to make the communication among processors explicit. Section 6 gives the scheduling mapping obtained from the wavefronts of computation and derives the program in which processors are scheduled to start computing at the earliest time step. Section 7 maps virtual processors onto physical processors to achieve perfect processor utilization. Section 8 summaries this paper and provides some concluding remarks.

2 The Language Crystal

The language we use to express the parallel programs is Crystal [2, 7]. Crystal is a strongly-typed, lexically scoped functional language with a simple syntax. Its syntax is based on the λ-calculus [1, 4], enriched with conditional and recursion. Its style is similar to many modern functional languages such as ML [11]. Appendix Appendix A provides some breif notes on the lambda notation. Let x_i range over identifiers, e_i and g_i range over expressions:

- A *program* is a set of mutually recursive definitions with an output expression:

$$x_1\texttt{=}e_1\texttt{,}\ x_2\texttt{=}e_2\texttt{,}\ \ldots\texttt{,}\ \texttt{?}e\ .$$

  The notation $x_i\texttt{=}e_i$ is a *definition* that binds the identifier x_i to the value of e_i. The result of the program is the value of the output expression $?e$. The order of definitions is irrelevant.

- The *conditional expressions* have the following form: "$e \rightarrow e_1, e_2$", where e is the predicate, e_1 is the consequence, and e_2 is the alternative. It is equivalent to "if e then e_1 else e_2". For nested conditional expressions $g_1 \rightarrow e_1, (g_2 \rightarrow e_2, (\ldots))$ we use the sugared form below for its clarity:

$$\left\{\begin{array}{l} g_1 \rightarrow e_1, \\ g_2 \rightarrow e_2, \\ \cdots \end{array}\right\}$$

- *Functions* are introduced by λ-abstraction and have the form $\lambda(x_1, x_2, \ldots).e$ where $x_1, x_2, \ldots$ are the formal arguments and e is the body of the function. Function *applications* have the form of $e_1(e_2, e_3, \ldots)$, where e_1 is the function and $e_2, e_2, \ldots$ are the arguments. When there is only one formal argument, we may omit the parenthesis around it.

- *Typed expressions* have the form $e_1 : e_2$, where e_2 should be a type expression. The type of the formal (i.e., the domain) of a function abstraction can be declared explicitly as well: $\lambda x\texttt{:}e_2\texttt{.}e_1$.

- An *expression* can be a constant (of type boolean, integer, float, and string), an identifier, a conditional expression, a function abstraction, or a function application.

Parenthesis can be used for grouping. All non-prefixed notations are considered as the sugared form of their equivalent prefixed function applications. For example, `1+2` is the sugared form of `add(1,2)`.

Parallel Interpretation of Crystal Programs

The major objects of interest in Crystal are data fields and index domains. A *data field* is a function over some index domain. An *index domain* is a set of index points. For example, $D_1 = \mathsf{interval}(1, m)$ is an interval index domain with m index points, indexed from 1 to m. Similarly for $D_2 = \mathsf{interval}(1, n)$. More complicated index domains can be constructed by cartesian product, coproduct, and restriction. The *cartesian product* of D_1 and D_2, denoted $D_1 \times D_2$, is an index domain with $m \times n$ index points, indexed from $(1, 1)$ to (m, n). The *coproduct* of D_1 and D_2, denoted $D_1 + D_2$, has $m + n$ points, indexed by $\iota_1(1),\ldots,\iota_1(m)$, and $\iota_2(1),\ldots,\iota_2(n)$, where $\iota_1 : D_1 \rightarrow (D_1 + D_2)$ and $\iota_2 : D_2 \rightarrow (D_1 + D_2)$ are the *injections*. The *restriction* of D_1 by a *filter* (i.e. a boolean function over domain

D_1) f, denoted $D_1 \,|\, f$, is the subdomain of D_1 in which only the index points passing (i.e. satisfying) the filter are included. For example, $D_3 = D_1 \,|\, (\lambda i.\mathsf{even?}(i))$ is a subdomain of D_1 where only points with even index are included. By definition, the formal argument of the filter (in this example, i) ranges over the domain to be restricted (in this example, D_1).

Data fields defined over index domains are to be distributed among processors and computed in parallel. Since data field definitions can be mutually recursive, ***data dependencies*** among index points of data fields are introduced by referencing the values of some data fields on some index points. These index domains eventually have to be distributed among processors of some target parallel machine. The data dependencies lead to the ***communication*** among processors if the involved index points are distributed to different processors. In order to minimize the communication cost, the compiler has to find a way to align (if the involved domains are of different size and dimension) and map the index domains to the processors in such a way that the resulting computation is efficient.

The description above is necessarily quite vague. Following sections will provide examples which will make the description clear. Interested readers are referred to [10, 8, 9] for Crystal's parallelizing compilation techinques.

3 Initial Program

Figure 1 shows the initial program of Gauss-Seidel. The domain N is an interval from 0 to n, where $n+1$ is the size of the mesh. The domain M is an interval from 0 to m, where m is the maximal iteration count. The domain D_1 is a product of the domain of mesh $N \times N$, or N^2 for short, and the iteration interval M. The data field A_0 over N^2 holds the initial mesh values, which will be read in from the input. The data field ϕ, indexed by (i,j,k), defines the iterative finite differencing. Indices (i,j) range over N^2 and the index k ranges over M. The value $\phi(i,j,k)$ is the value on the mesh point (i,j) at the k-th iteration. The first four clauses of the condition in the definition of ϕ, $i=0$ or $i=n$ or $j=0$ or $j=n$, test for the boundary condition. The last clause, $k=0$, tests for the initial condition. When either the boundary condition or the initial condition holds, $\phi(i,j,k)$ is defined to be $A_0(i,j)$, the initial mesh value at (i,j). Otherwise, $\phi(i,j,k)$ is defined to be $(\phi(i-1,j,k)+\phi(i,j-1,k)+\phi(i+1,j,k-1)+\phi(i,j+1,k-1))/4$, which is exactly the definition of Gauss-Seidel. The datafield A is the final mesh value. The correctness of the program p1 can be established easily by structural induction over the domain D_1.

$$
\begin{aligned}
N &= \mathsf{interval}(0,n) \\
M &= \mathsf{interval}(0,m) \\
D_1 &= N \times N \times M \\
A_0 &= \lambda(i,j) : N^2.\ \text{initial mesh values, read from input} \\
\phi &= \lambda(i,j,k) : D_1. \left\{ \begin{array}{l} i=0 \ \mathsf{or}\ i=n \ \mathsf{or}\ j=0 \ \mathsf{or}\ j=n \ \mathsf{or}\ k=0 \rightarrow A_0(i,j) \\ \mathsf{else}\ \rightarrow (\phi(i-1,j,k)+\phi(i,j-1,k)+\phi(i+1,j,k-1)+\phi(i,j+1,k-1))/4 \end{array} \right\} \\
A &= \lambda(i,j) : N^2.\ \phi(i,j,m) \\
&?\ A
\end{aligned}
$$

Figure 1: Program p1, the initial Crystal program for Gauss-Seidel method.

4 Partitioning the Problem Domain

We can explicitly decompose the two-dimensional mesh N^2 in the program p1 onto a two-dimensional mesh of processors, each with its own two-dimensional local partition, with the following domains and mappings:

$$
\begin{aligned}
P &= \mathsf{interval}(0,(n+1)/b-1) \\
L &= \mathsf{interval}(0,b-1) \\
D_2 &= P \times P \times M \times L \times L \\
\pi &= \lambda(i,j,k) : D_1.(i \ \mathsf{div}\ b, j \ \mathsf{div}\ b, k, i \ \mathsf{mod}\ b, j \ \mathsf{mod}\ b) : D_2 \\
\pi^{-1} &= \lambda(x,y,k,u,v) : D_2.(x*b+u, y*b+v, k) : D_1
\end{aligned}
$$

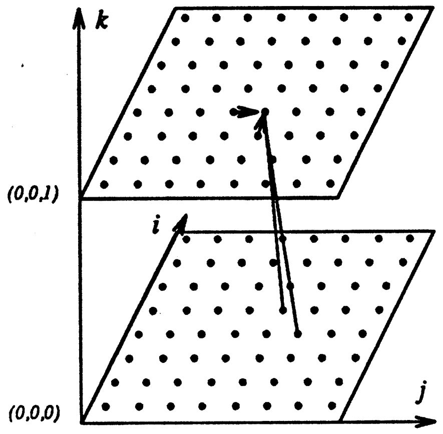

Figure 2: Gauss-Seidel on a two-dimensional mesh.

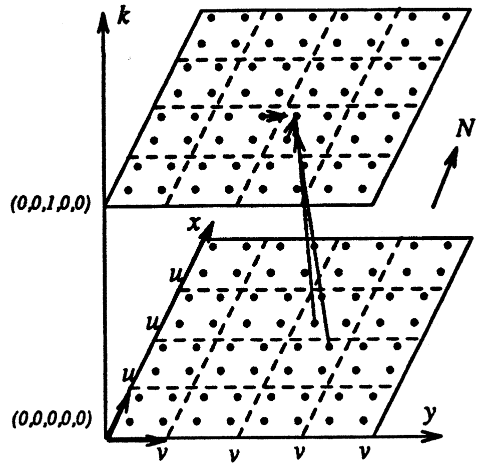

Figure 3: Partitioned Gauss-Seidel on a two-dimensional mesh of processors, each has its own local partition.

where the operator div is the integer division and the operator mod is the integer modulo. The domain P is an interval from 0 to $(n+1)/b-1$, where b is the block size of each local partition. P^2 is the domain of *processor* index. The domain L is an interval from 0 to $b-1$. L^2 is the domain of *local* mesh points. The partition morphism π maps $(i,j,k) \in D_1$ (i.e., the mesh point (i,j) at the k-th iteration) to the point $(x,y,k,u,v) \in D_2$ (i.e., the *local* mesh point (u,v) on the processor (x,y) at the k-th iteration). The morphism π^{-1} is the inverse of π.

Let V be the domain of floating point numbers and let D_2, π and π^{-1} be as defined above. We would like to derive a new data field $\varphi : D_2 \rightarrow V$ from the data field $\phi : D_1 \rightarrow V$ in the program p1 such that the following diagram commutes:

$$\begin{tikzcd}
D_1 \arrow[r, "\phi"] \arrow[d, shift right, "\pi"'] & V \\
D_2 \arrow[u, shift right, "\pi^{-1}"'] \arrow[ur, "\varphi"'] &
\end{tikzcd}$$

We know that $\phi = \varphi \circ \pi$ and $\varphi = \phi \circ \pi^{-1}$, where $\circ$ is the composition operator, are valid equations by the definition of φ. The derivation for φ is as follows:

$$\begin{aligned}
\varphi &= \phi \circ \pi^{-1} \qquad \text{by definition} \\
&= \lambda(x,y,k,u,v) : D_2.\ ((\phi \circ \pi^{-1})(x,y,k,u,v)) \qquad \text{– by } \eta\text{-abstraction} \\
&= \lambda(x,y,k,u,v) : D_2.\ \phi(\pi^{-1}(x,y,k,u,v)) \qquad \text{– by unfolding composition} \\
&= \lambda(x,y,k,u,v) : D_2.\ \phi(x*b+u, y*b+v, k) \qquad \text{– by unfolding } \pi^{-1} \\
&= \lambda(x,y,k,u,v) : D_2. \\
&\quad \left\{ \begin{array}{l} ((x*b+u=0 \text{ or } x*b+u=n \text{ or } y*b+v=0 \text{ or } y*b+v=n) \text{ or } k=1) \\ \quad \rightarrow A(x*b+u, y*b+v) \\ \text{else } \rightarrow (\phi(x*b+u-1, y*b+v, k) + \phi(x*b+u, y*b+v-1, k) + \\ \quad \phi(x*b+u+1, y*b+v, k-1) + \phi(x*b+u, y*b+v+1, k-1))/4 \end{array} \right\} \\
&\qquad\qquad \text{– by unfolding } \phi \\
&= \lambda(x,y,k,u,v) : D_2.
\end{aligned}$$

$$\left\{\begin{array}{l} ((x*b+u=0 \text{ or } x*b+u=n \text{ or } y*b+v=0 \text{ or } y*b+v=n) \text{ or } k=1) \\ \quad \rightarrow A(x*b+u, y*b+v) \\ \text{else } \rightarrow ((\varphi\circ\pi)(x*b+u-1, y*b+v, k)+(\varphi\circ\pi)(x*b+u, y*b+v-1, k)+ \\ \quad (\varphi\circ\pi)(x*b+u+1, y*b+v, k-1)+(\varphi\circ\pi)(x*b+u, y*b+v+1, k-1))/4 \end{array}\right\}$$

– by substituting ϕ with $\varphi\circ\pi$

$$= \lambda(x,y,k,u,v) : D_2.$$

$$\left\{\begin{array}{l} (x*b+u=0 \text{ or } x*b+u=n \text{ or } y*b+v=0 \text{ or } y*b+v=n) \\ \quad \text{or } k=1 \rightarrow A(x*b+u, y*b+v) \\ \text{else } \rightarrow (\varphi((x*b+u-1) \text{ div } b, y, k, (x*b+u-1) \text{ mod } b, v)+ \\ \qquad \varphi(x, (y*b+v-1) \text{ div } b, k, u, (x*b+u-1) \text{ mod } b)+ \\ \qquad \varphi(x*b+u+1) \text{ div } b, y, k-1, (x*b+u+1) \text{ mod } b, v)+ \\ \qquad \varphi(x, (y*b+v+1) \text{ div } b, k-1, u, (x*b+u+1) \text{ mod } b))/4 \end{array}\right\}$$

– by unfolding composition, then unfolding π

Now we have the data field φ defined over explicit processors and local mesh points. The value $\varphi(x,y,k,u,v)$ means the mesh value on processor (x,y) at local mesh point (u,v) at the k-th iteration. The transformation technique we used above is called *reshape* since the underlying index domain D_1 is reshaped into a new index domain D_2 by the one-to-one function π.

We can also partition the initial mesh values onto the mesh of processors by introducing the following new definition:

$$\bar{A}_0 = \lambda(x,y,,u,v) : (P\times P\times L\times L).\ A_0(x*b+u, y+b+v)$$

By substituting $A(x*b+u, y*b+v)$ with $\bar{A}_0(x,y,u,v)$, which is valid by the definition of $\bar{A}_0$, in the last definition of φ above, we have the final definition of φ as shown in Figure 4. Similarly, we can derive another equivalent version of A that is defined in terms of φ as follows:

$$\begin{array}{rcll} A &=& \lambda(i,j) : N^2.\ \phi(i,j,m) & \\ &=& \lambda(i,j) : N^2.\ (\varphi\circ\pi)(i,j,m) & \text{– by substituting } \phi \text{ with } \varphi\circ\pi \\ &=& \lambda(i,j) : N^2.\ \varphi(i \text{ div } b, j \text{ div } b, m, i \text{ mod } b, j \text{ mod } b) & \text{– by unfolding composition, then unfolding } \pi \end{array}$$

We have derived a new program **p2** as shown in Figure 4. It has the problem domain explicitly decomposed among a mesh of processors. The equivalence between the programs **p1** and **p2** is based on the equivalence of the two definitions of A, which has been established by the validness of our derivation above.

5 Coordinating the Communication

Communication between processors happens when the value of φ on the index point (x,y,k,u,v) depends on some other index point with different processor indices, that is, depends on some index point (x',y',k',u',v') where $x\neq x'$ or $y\neq y'$. From the definition of φ in p2, $\varphi(x,y,k,u,v)$ is dependent on the following four index points

$$\begin{array}{l} ((x*b+u-1) \text{ div } b, y, k, (x*b+u-1) \text{ mod } b, v) \\ (x, (y*b+v-1) \text{ div } b, k, u, (x*b+u-1) \text{ mod } b) \\ ((x*b+u+1) \text{ div } b, y, k-1, (x*b+u+1) \text{ mod } b, v) \\ (x, (y*b+v+1) \text{ div } b, k-1, u, (x*b+u+1) \text{ mod } b) \end{array}$$

when the condition $x*b+u=0$ or $x*b+u=n$ or $y*b+v=0$ or $y*b+v=n$ or $k=1$ is false. We need to analyze the range of the first two indices of these four points to decide when they will yield values different from x and y, that is, when they will reside in another processor other than the processor (x,y). Since (x,y) range over P^2 and

$$
\begin{aligned}
N &= \mathsf{interval}(0, n)\\
M &= \mathsf{interval}(0, m)\\
A_0 &\coloneqq \lambda(i,j) : N^2.\ \text{initial mesh values, read from input}\\
L &= \mathsf{interval}(0, b-1)\\
P &= \mathsf{interval}(0, (n+1)/b - 1)\\
D_2 &= P \times P \times M \times L \times L\\
\bar{A}_0 &= \lambda(x,y,u,v) : (P \times P \times L \times L).\ A_0(x*b+u, y*b+v)\\
\varphi &= \lambda(x,y,k,u,v) : D_2. \left\{ \begin{array}{l}
(x*b+u = 0 \text{ or } x*b+u = n \text{ or } y*b+v = 0 \text{ or } y*b+v = n)\\
\quad \text{or } k = 0 \rightarrow \bar{A}_0(x,y,u,v)\\
\text{else } \rightarrow (\varphi((x*b+u-1) \text{ div } b, y, k, (x*b+u-1) \text{ mod } b, v) +\\
\qquad \varphi(x, (y*b+v-1) \text{ div } b, k, u, (x*b+u-1) \text{ mod } b) +\\
\qquad \varphi((x*b+u+1) \text{ div } b, y, k-1, (x*b+u+1) \text{ mod } b, v) +\\
\qquad \varphi(x, (y*b+v+1) \text{ div } b, k-1, u, (x*b+u+1) \text{ mod } b))/4
\end{array} \right\}\\
A &= \lambda(i,j) : N^2.\ \varphi(i \text{ div } b, j \text{ div } b, m, i \text{ mod } b, j \text{ mod } b)\\
?\ & A
\end{aligned}
$$

Figure 4: Program p2, partitioned Gauss-Seidel on a mesh of processors with explicit processors, but inexplicit communication.

(u, v) range over L^2, we know the followings:

$$
u = 0 \Rightarrow \left\{ \begin{array}{l} (x*b+u-1) \text{ div } b = x-1,\\ (x*b+u-1) \text{ mod } b = b-1. \end{array} \right.
$$

$$
0 < u < b-1 \Rightarrow \left\{ \begin{array}{l} (x*b+u-1) \text{ div } b = (x*b+u+1) \text{ div } b = x,\\ (x*b+u-1) \text{ mod } b = u-1,\\ (x*b+u+1) \text{ mod } b = u+1. \end{array} \right.
$$

$$
u = b-1 \Rightarrow \left\{ \begin{array}{l} (x*b+u+1) \text{ div } b = x+1,\\ (x*b+u+1) \text{ mod } b = 0. \end{array} \right.
$$

$$
v = 0 \Rightarrow \left\{ \begin{array}{l} (y*b+v-1) \text{ div } b = y-1,\\ (y*b+v-1) \text{ mod } b = b-1. \end{array} \right.
$$

$$
0 < v < b-1 \Rightarrow \left\{ \begin{array}{l} (y*b+v-1) \text{ div } b = (y*b+v+1) \text{ div } b = y,\\ (y*b+v-1) \text{ mod } b = v-1,\\ (y*b+v+1) \text{ mod } b = v+1. \end{array} \right.
$$

$$
v = b-1 \Rightarrow \left\{ \begin{array}{l} (y*b+v+1) \text{ div } b = y+1,\\ (y*b+v+1) \text{ mod } b = 0. \end{array} \right.
$$

By breaking the **else** clause in the definition of φ in **p2** into the nine cases resulting from combinations of three different possible ranges of u and v, we have the following derivation of φ:

$$\varphi = \lambda(x,y,k,u,v) : D_2.$$

$$\left\{\begin{array}{l}
(x*b+u=0 \text{ or } x*b+u=n \text{ or } y*b+v=0 \text{ or } y*b+v=n) \\
\quad \text{or } k=1 \rightarrow \bar{A}_0(x,y,u,v) \\
\text{else} \rightarrow \left\{\begin{array}{l}
u=0 \text{ and } v=0 \rightarrow (\varphi(x-1,y,k,b-1,v)+\varphi(x,y-1,k,u,b-1)+ \\
\quad \varphi(x,y,k-1,u+1,v)+\varphi(x,y,k-1,u,v+1))/4 \\
u=0 \text{ and } 0<v<b-1 \rightarrow (\varphi(x-1,y,k,b-1,v)+\varphi(x,y,k,u,v-1)+ \\
\quad \varphi(x,y,k-1,u+1,v)+\varphi(x,y,k-1,u,v+1))/4 \\
u=0 \text{ and } v=b-1 \rightarrow (\varphi(x-1,y,k,b-1,v)+\varphi(x,y,k,u,v-1)+ \\
\quad \varphi(x,y,k-1,u+1,v)+\varphi(x,y+1,k-1,u,0))/4 \\
0<u<b-1 \text{ and } v=0 \rightarrow (\varphi(x,y,k,u-1,v)+\varphi(x,y-1,k,u,b-1)+ \\
\quad \varphi(x,y,k-1,u+1,v)+\varphi(x,y,k-1,u,v+1))/4 \\
0<u<b-1 \text{ and } 0<v<b-1 \rightarrow (\varphi(x,y,k,u-1,v)+\varphi(x,y,k,u,v-1)+ \\
\quad \varphi(x,y,k-1,u+1,v)+\varphi(x,y,k-1,u,v+1))/4 \\
0<u<b-1 \text{ and } v=b-1 \rightarrow (\varphi(x,y,k,u-1,v)+\varphi(x,y,k,u,v-1)+ \\
\quad \varphi(x,y,k-1,u+1,v)+\varphi(x,y+1,k-1,u,0))/4 \\
u=b-1 \text{ and } v=0 \rightarrow (\varphi(x,y,k,u-1,v)+\varphi(x,y-1,k,u,b-1)+ \\
\quad \varphi(x+1,y,k-1,0,v)+\varphi(x,y,k-1,u,v+1))/4 \\
u=b-1 \text{ and } 0<v<b-1 \rightarrow (\varphi(x,y,k,u-1,v)+\varphi(x,y,k,u,v-1)+ \\
\quad \varphi(x+1,y,k-1,0,v)+\varphi(x,y,k-1,u,v+1))/4 \\
u=b-1 \text{ and } v=b-1 \rightarrow (\varphi(x,y,k,u-1,v)+\varphi(x,y,k,u,v-1)+ \\
\quad \varphi(x+1,y,k-1,0,v)+\varphi(x,y+1,k-1,u,0))/4
\end{array}\right\}
\end{array}\right\}$$

In order to factor out the interprocessor communication, we introduce the following four new definitions :

$$\begin{aligned}
\mathcal{N} &= \lambda(x,y,k,u,v) : D_2.\ \varphi(x+1,y,k-1,0,v) \\
\mathcal{E} &= \lambda(x,y,k,u,v) : D_2.\ \varphi(x,y+1,k-1,u,0) \\
\mathcal{W} &= \lambda(x,y,k,u,v) : D_2.\ \varphi(x,y-1,k,u,b-1) \\
\mathcal{S} &= \lambda(x,y,k,u,v) : D_2.\ \varphi(x-1,y,k,b-1,v)
\end{aligned}$$

where $\mathcal{N}$, $\mathcal{E}, \mathcal{W}$, and $\mathcal{S}$ holds values from the northern, eastern, western, and southern neighbors, respectively. We choose the orientation of north to be the direction of positive x (and positive u) as in Figure 4. By substituting $\varphi(x+1,y,k-1,0,v)$ with $\mathcal{N}(x,y,k,u,v)$, $\varphi(x,y+1,k-1,u,0)$ with $\mathcal{E}(x,y,k,u,v)$, $\varphi(x,y-1,k,u,b-1)$ with $\mathcal{W}(x,y,k,u,v)$, and $\varphi(x-1,y,k,b-1,v)$ with $\mathcal{S}(x,y,k,u,v)$ (which are all valid equalities derived from the definitions above) we have the new definition of φ in the program p3, as shown in Figure 5. Now the values of the data field φ depend only on values in the same processor, where the communication between processors are handled by four explicit communication buffers specified by $\mathcal{N}, \mathcal{E}, \mathcal{W}$ and $\mathcal{S}$.

6 Scheduling Processors with Wavefronts

A naive implementation of the program p3 by using the index k as time step will not have efficient processor utilization because, at every k, processors have to wait for values propagated from its western and southern neighbors due to the data dependency. This implies that at any instant we have at most p processors active even though we have p^2 processors available, where p is the size of the target processor mesh. With closer examination on the data dependencies among the processors, we observe that the computation can proceed in a wavefront fashion as shown in Figure 6. The numbering on the partitions indicates the *earliest* relative time steps when the computation on that partition can proceed. These wavefronts suggest a good time index along which processors are scheduled to start at the earliest possible time step. Consider the following domains and linear mappings:

$$
\begin{aligned}
N &= \mathsf{interval}(0,n)\\
M &= \mathsf{interval}(0,m)\\
A_0 &= \lambda(i,j) : N^2.\ \text{initial mesh values, read from input}\\
L &= \mathsf{interval}(0,b-1)\\
P &= \mathsf{interval}(0,(n+1)/b-1)\\
D_2 &= P \times P \times M \times L \times L\\
\bar{A}_0 &= \lambda(x,y,u,v) : (P \times P \times L \times L).\ A_0(x*b+u, y*b+v)\\
\mathcal{N} &= \lambda(x,y,k,u,v) : D_2.\ \varphi(x+1,y,k-1,0,v)\\
\mathcal{E} &= \lambda(x,y,k,u,v) : D_2.\ \varphi(x,y+1,k-1,u,0)\\
\mathcal{W} &= \lambda(x,y,k,u,v) : D_2.\ \varphi(x,y-1,k,u,b-1)\\
\mathcal{S} &= \lambda(x,y,k,u,v) : D_2.\ \varphi(x-1,y,k,b-1,v)\\
\varphi &= \lambda(x,y,k,u,v) : D_2.\\
&\left\{\begin{array}{l}
(x*b+u=0 \text{ or } x*b+u=n \text{ or } y*b+v=0 \text{ or } y*b+v=n)\\
\quad \text{or } k=0 \rightarrow \bar{A}_0(x,y,u,v)\\
\text{else} \rightarrow \left\{\begin{array}{l}
u=0 \text{ and } v=0 \rightarrow (\mathcal{S}(x,y,k,u,v)+\mathcal{W}(x,y,k,u,v)+\\
\qquad \varphi(x,y,k-1,u+1,v)+\varphi(x,y,k-1,u,v+1))/4\\
u=0 \text{ and } 0<v<b-1 \rightarrow (\mathcal{S}(x,y,k,u,v)+\varphi(x,y,k,u,v-1)+\\
\qquad \varphi(x,y,k-1,u+1,v)+\varphi(x,y,k-1,u,v+1))/4\\
u=0 \text{ and } v=b-1 \rightarrow (\mathcal{S}(x,y,k,u,v)+\varphi(x,y,k,u,v-1)+\\
\qquad \varphi(x,y,k-1,u+1,v)+\mathcal{E}(x,y,k,u,v))/4\\
0<u<b-1 \text{ and } v=0 \rightarrow (\varphi(x,y,k,u-1,v)+\mathcal{W}(x,y,k,u,v)+\\
\qquad \varphi(x,y,k-1,u+1,v)+\varphi(x,y,k-1,u,v+1))/4\\
0<u<b-1 \text{ and } 0<v<b-1 \rightarrow (\varphi(x,y,k,u-1,v)+\varphi(x,y,k,u,v-1)+\\
\qquad \varphi(x,y,k-1,u+1,v)+\varphi(x,y,k-1,u,v+1))/4\\
0<u<b-1 \text{ and } v=b-1 \rightarrow (\varphi(x,y,k,u-1,v)+\varphi(x,y,k,u,v-1)+\\
\qquad \varphi(x,y,k-1,u+1,v)+\mathcal{E}(x,y,k,u,v))/4\\
u=b-1 \text{ and } v=0 \rightarrow (\varphi(x,y,k,u-1,v)+\mathcal{W}(x,y,k,u,v)+\\
\qquad \mathcal{N}(x,y,k,u,v)+\varphi(x,y,k-1,u,v+1))/4\\
u=b-1 \text{ and } 0<v<b-1 \rightarrow (\varphi(x,y,k,u-1,v)+\varphi(x,y,k,u,v-1)+\\
\qquad \mathcal{N}(x,y,k,u,v)+\varphi(x,y,k-1,u,v+1))/4\\
u=b-1 \text{ and } v=b-1 \rightarrow (\varphi(x,y,k,u-1,v)+\varphi(x,y,k,u,v-1)+\\
\qquad \mathcal{N}(x,y,k,u,v)+\mathcal{E}(x;y,k,u,v))/4
\end{array}\right\}
\end{array}\right\}\\
A &= \lambda(i,j) : N^2.\ \varphi(i \,\mathsf{div}\, b, j \,\mathsf{div}\, b, m, i \,\mathsf{mod}\, b, j \,\mathsf{mod}\, b)\\
?\ A &
\end{aligned}
$$

Figure 5: Program p3, partitioned Gauss-Seidel on a mesh of processors with explicit communication buffers.

$$
\begin{aligned}
T &= \mathsf{interval}(0, 2n+m)\\
D_3 &= (P \times P \times T \times L \times L)|(\lambda(p,q,t,u,v).(p,q,(t-p-q)/2,u,v) \text{ in? } D_2)\\
g &= \lambda(x,y,k,u,v) : D_2.(x,y,x+y+2k,u,v) : D_3\\
g^{-1} &= \lambda(p,q,t,\bar{u},\bar{v}) : D_3.(p,q,(t-p-q)/2,\bar{u},\bar{v}) : D_2
\end{aligned}
$$

The mapping g maps points in D_2 to processors in D_3, which has an explicit time domain T, and g^{-1} does the inverse. The time component of g, i.e., $x+y+2k$, is obtained from the plane equations of the wavefronts, which can

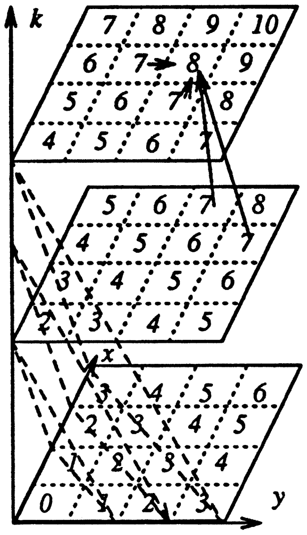

Figure 6: Wavefronts of computation of Gauss-Seidel that are used to schedule the processors.

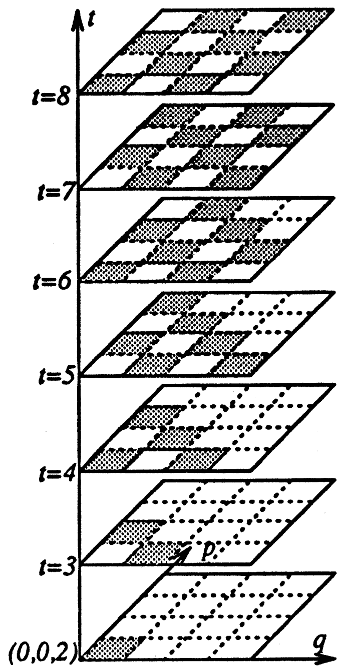

Figure 7: Gauss-Seidel with explicit processor scheduling. Shaded area indicating processor active.

be found by simple analytical geometry. Given g, the domain T can be derived from mapping the extreme points of D_2 through g. The domain D_3 is the domain $P \times P \times T \times L \times L$ restricted by a predicate which filters out extra points so that D_2 and D_3 have the same cardinality. The mapping g^{-1} is derived by solving the simple linear equation system: $x = p, y = q, x + y + 2k = t, u = \bar{u}, v = \bar{v}$. We would like to derive new data fields $\psi, \bar{\mathcal{N}}, \bar{\mathcal{E}}, \bar{\mathcal{W}}$ and $\bar{\mathcal{S}}$ defined over D_3, such that the following diagram commutes:

$$\begin{tikzcd} D_2 \arrow[r, "\varphi{,}\mathcal{N}"] \arrow[d, shift right, "g"'] & V \\ D_3 \arrow[u, shift right, "g^{-1}"'] \arrow[ur, "\psi{,}\bar{\mathcal{N}}{,}\text{ etc.}"'] & \end{tikzcd}$$

Using the *reshape* transformation technique used in Section 4, we can derive $\bar{\mathcal{N}}$ as follows:

$$\begin{aligned}
\bar{\mathcal{N}} &= \mathcal{N} \circ g^{-1} && \text{– by definition}\\
&= \lambda(p,q,t,u,v) : D_3.\ ((\mathcal{N} \circ g^{-1})(p,q,t,u,v)) && \text{– by } \eta\text{-abstraction}\\
&= \lambda(p,q,t,u,v) : D_3.\ \mathcal{N}(g^{-1}(p,q,t,u,v)) && \text{– by unfolding composition}\\
&= \lambda(p,q,t,u,v) : D_3.\ \mathcal{N}(p,q,(t-p-q)/2,u,v) && \text{– by unfolding } g^{-1}\\
&= \lambda(p,q,t,u,v) : D_3.\ \varphi(p+1,q,(t-p-q)/2-1,0,v) && \text{– by unfolding } \mathcal{N}\\
&= \lambda(p,q,t,u,v) : D_3.\ (\psi \circ g)(p+1,q,(t-p-q)/2-1,0,v) && \text{– by substituting } \varphi \text{ with } \psi \circ g\\
&= \lambda(p,q,t,u,v) : D_3.\ \psi(p+1,q,t-1,0,v) && \text{– by unfolding composition, then unfolding } g
\end{aligned}$$

Similarly, we can derive $\bar{\mathcal{E}}, \bar{\mathcal{W}}, \bar{\mathcal{S}}$ and ψ. Notice that the derivation of ψ needs a few more steps to replace the occurrences of $\mathcal{N}, \mathcal{E}, \mathcal{W}$ and $\mathcal{S}$ with $\bar{N} \circ g, \bar{E} \circ g, \bar{W} \circ g$ and $\bar{S} \circ g$, respectively, followed by unfolding the composition and g, so that ψ depends only on the new data fields defined over D_3. We also need to replace the occurrence of φ in the definition of A with $\psi \circ g$, then unfold the composition and g, in order to get an equivalent definition of A that depends on ψ. The result of these derivations is the program **p4**, as shown in Figure 8.

Figure 7 shows the effect of the scheduling according to wavefronts. The period of $2 \le t \le 6$ is the latency to fill up the pipeline. After $t = 7$, processors proceed in a checker-board alternation fashion. At every time step, half of the available processors are active. Therefore, we have achieved a much better processor utilization. Notice that during the pipeline latency, processors need access to initial mesh values, which is handled properly by the new initial condition $t = p + q$ in the definition of ψ in p4.

$$
\begin{aligned}
N &= \mathsf{interval}(0, n)\\
A_0 &= \lambda(i,j) : N^2.\ \text{initial mesh values, read from input}\\
L &= \mathsf{interval}(0, b-1)\\
P &= \mathsf{interval}(0, (n+1)/b-1)\\
T &= \mathsf{interval}(0, 2n+m)\\
D_3 &= P \times P \times T \times L \times L\\
\bar{A}_0 &= \lambda(x,y,u,v) : (P \times P \times L \times L).\ A_0(x*b+u, y*b+v)\\
\bar{\mathcal{N}} &= \lambda(p,q,t,u,v) : D_3.\ \psi(p+1, q, t-1, 0, v)\\
\bar{\mathcal{E}} &= \lambda(p,q,t,u,v) : D_3.\ \psi(p, q+1, t-1, u, 0)\\
\bar{\mathcal{W}} &= \lambda(p,q,t,u,v) : D_3.\ \psi(p, q-1, t-1, u, b-1)\\
\bar{\mathcal{S}} &= \lambda(p,q,t,u,v) : D_3.\ \psi(p-1, q, t-1, b-1, v)\\
\psi &= \lambda(p,q,t,u,v) : D_3.\\
&\left\{\begin{array}{l}
(p*b+u=0 \text{ or } p*b+u=n \text{ or } q*b+v=0 \text{ or } q*b+v=n)\\
\quad \text{or } t=p+q \rightarrow \bar{A}_0(p,q,u,v)\\
\text{else} \rightarrow \left\{\begin{array}{l}
u=0 \text{ and } v=0 \rightarrow (\bar{\mathcal{S}}(p,q,t,u,v)+\bar{\mathcal{W}}(p,q,t,u,v)+\\
\qquad \psi(p,q,t-2,u+1,v)+\psi(p,q,t-2,u,v+1))/4\\
u=0 \text{ and } 0<v<b-1 \rightarrow (\bar{\mathcal{S}}(p,q,t,u,v)+\psi(p,q,t,u,v-1)+\\
\qquad \psi(p,q,t-2,u+1,v)+\psi(p,q,t-2,u,v+1))/4\\
u=0 \text{ and } v=b-1 \rightarrow (\bar{\mathcal{S}}(p,q,t,u,v)+\psi(p,q,t,u,v-1)+\\
\qquad \psi(p,q,t-2,u+1,v)+\bar{\mathcal{E}}(p,q,t,u,v))/4\\
0<u<b-1 \text{ and } v=0 \rightarrow (\psi(p,q,t,u-1,v)+\bar{\mathcal{W}}(p,q,t,u,v)+\\
\qquad \psi(p,q,t-2,u+1,v)+\psi(p,q,t-2,u,v+1))/4\\
0<u<b-1 \text{ and } 0<v<b-1 \rightarrow (\psi(p,q,t,u-1,v)+\psi(p,q,t,u,v-1)+\\
\qquad \psi(p,q,t-2,u+1,v)+\psi(p,q,t-2,u,v+1))/4\\
0<u<b-1 \text{ and } v=b-1 \rightarrow (\psi(p,q,t,u-1,v)+\psi(p,q,t,u,v-1)+\\
\qquad \psi(p,q,t-2,u+1,v)+\bar{\mathcal{E}}(p,q,t,u,v))/4\\
u=b-1 \text{ and } v=0 \rightarrow (\psi(p,q,t,u-1,v)+\bar{\mathcal{W}}(p,q,t,u,v)+\\
\qquad \bar{\mathcal{N}}(p,q,t,u,v)+\psi(p,q,t-2,u,v+1))/4\\
u=b-1 \text{ and } 0<v<b-1 \rightarrow (\psi(p,q,t,u-1,v)+\psi(p,q,t,u,v-1)+\\
\qquad \bar{\mathcal{N}}(p,q,t,u,v)+\psi(p,q,t-2,u,v+1))/4\\
u=b-1 \text{ and } v=b-1 \rightarrow (\psi(p,q,t,u-1,v)+\psi(p,q,t,u,v-1)+\\
\qquad \bar{\mathcal{N}}(p,q,t,u,v)+\bar{\mathcal{E}}(p,q,t,u,v))/4
\end{array}\right\}
\end{array}\right\}\\
A &= \lambda(i,j) : N^2.\ \psi(i \text{ div } b, j \text{ div } b, (i \text{ div } b)+(j \text{ div } b)+2m, i \bmod b, j \bmod b)\\
?\ & A
\end{aligned}
$$

Figure 8: Program p4, partitioned Gauss-Seidel on a mesh of processors with explicit time index.

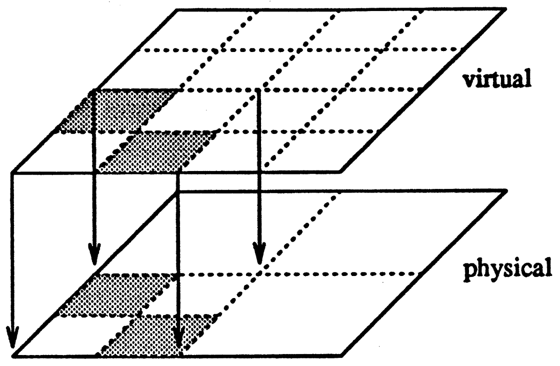

Figure 9: One physical processor emulates four virtual processors to achieve perfect processor utilization.

7 Achieving Perfect Processor Utilization

In the program p4, only half of the total processors are active at every instant and the other half are left idle, which is wasteful. We can further improve the actual processor utilization by having one *physical* processor emulate four, a two by two square as shown in Figure 9, *virtual* processors in p4 so that every physical processor is fully utilized through the course of computation. One way to achieve this goal is by the following domains and mappings:

$$
\begin{aligned}
L_2 &= \text{interval}(0, 2b-1) \\
P_{\frac{1}{2}} &= \text{interval}(0, \frac{n+1}{2b} - 1) \\
D_4 &= P_{\frac{1}{2}} \times P_{\frac{1}{2}} \times T \times L_2 \times L_2 \\
h &= \lambda(p,q,t,u,v) : D_3. \left\{ \begin{array}{l} \text{even?}(p) \text{ and even?}(q) \rightarrow (p \text{ div } 2, q \text{ div } 2, t, u, v) \\ \text{even?}(p) \text{ and odd?}(q) \rightarrow (p \text{ div } 2, q \text{ div } 2, t, u, v+b) \\ \text{odd?}(p) \text{ and even?}(q) \rightarrow (p \text{ div } 2, q \text{ div } 2, t, u+b, v) \\ \text{odd?}(p) \text{ and odd?}(q) \rightarrow (p \text{ div } 2, q \text{ div } 2, t, u+b, v+b) \end{array} \right\} : D_4 \\
h^{-1} &= \lambda(\hat{p},\hat{q},t,\hat{u},\hat{v}) : D_4. \left\{ \begin{array}{l} \hat{u} < b \text{ and } \hat{v} < b \rightarrow (2\hat{p}, 2\hat{q}, t, \hat{u}, \hat{v}) \\ \hat{u} < b \text{ and } \hat{v} \geq b \rightarrow (2\hat{p}, 2\hat{q}+1, t, \hat{u}, \hat{v}-b) \\ \hat{u} \geq b \text{ and } \hat{v} < b \rightarrow (2\hat{p}+1, 2\hat{q}, t, \hat{u}-b, \hat{v}) \\ \hat{u} \geq b \text{ and } \hat{v} \geq b \rightarrow (2\hat{p}+1, 2\hat{q}+1, t, \hat{u}-b, \hat{v}-b) \end{array} \right\} : D_3
\end{aligned}
$$

The domain $L_2 \times L_2$ is the actual partition on each physical processor. It is four times as large as $L \times L$. The domain $P_{\frac{1}{2}} \times P_{\frac{1}{2}}$ is the domain of physical processors. It is only one quarter of $P \times P$. Therefore we have one physical processor in D_4 emulate four virtual processors in D_3. The mapping h maps points in D_3 into points in D_4. It maps the virtual processor index (p,q) and virtual local mesh point (u,v) into the physical processor index and physical local mesh according to which quadrant the virtual processor falls in the actual local partition. The mapping h^{-1} does the inverse. Again, we need derive new data fields $\hat{\psi}, \hat{\mathcal{N}}$, etc, that defined over D_4 so that the following diagram commutes.

$$\begin{tikzcd}
D_3 \arrow[r, "\psi{,}\mathcal{N}"] \arrow[d, shift right, "h"'] & V \\
D_4 \arrow[u, shift right, "h^{-1}"'] \arrow[ur, "\hat{\psi}{,}\hat{\mathcal{N}}{,}\text{ etc.}"']
\end{tikzcd}$$

Reshape applies again but with a slight new twist since now the h and h^{-1} are defined with conditionals instead of simple mappings as in Section 4 and 6. What we need is the following axiom that allows us to distribute application over conditionals. Let f be a function, g_i be boolean expressions, and e_i be expressions, the following is true:

$$f(\left\{\begin{array}{l} g_1 \rightarrow e_1 \\ g_2 \rightarrow e_2 \\ \dots \end{array}\right\}) = \left\{\begin{array}{l} g_1 \rightarrow f(e_1) \\ g_2 \rightarrow f(e_2) \\ \dots \end{array}\right\}$$

Then we will be able to derive a new program defined over D_4 which is highly efficient in both processor scheduling and processor utilization.

8 Summaries and Concluding Remarks

We have presented a formal derivation of a highly efficient parallel Gauss-Seidel method for parallel machines based on two-dimensional mesh of processors. Our approach is summarized in the following diagram:

$$\begin{tikzcd}
D_1 \arrow[r, shift left, "\pi"] \arrow[drr, "\mathbf{p1}"'] & D_2 \arrow[l, shift left, "\pi^{-1}"] \arrow[r, shift left, "g"] \arrow[dr, "\mathbf{p2}"] & D_3 \arrow[l, shift left, "g^{-1}"] \arrow[r, shift left, "h"] \arrow[d, "\mathbf{p4}"] & D_4 \arrow[l, shift left, "h^{-1}"] \arrow[dl, "\mathbf{p5}"] \\
 & & V &
\end{tikzcd}$$

We start from the initial program **p1** defined over the logical problem domain D_1, which is abstract from any implementation details. Then we use a pair of bijective mappings π and π^{-1} to transform D_1 into a more concrete index domain D_2 which has the information about processors and local partitions on each processor. An equational transformation technique *reshape* is used to derive the second program **p2** defined over D_2. Dependency analysis is then done to make the communication between processors explicit and a new program **p3** is derived algebraically. A second pair of bijective mappings g and g^{-1} is then used to reshape D_2 into D_3 which schedules processors to start computation at the earliest time step with an explicit time index according to computation wavefronts in **p3**. Reshape is used again to derive the new program **p4** defined over D_3. Finally, a third pair of bijective mappings h and h^{-1} are used to transformed D_3 into D_4 by making every physical processor in D_4 emulate four virtual processors in D_3 so that the processor utilization is perfect. Again, reshape can be used to derive the final program.

We believe this transformational framework is suitable for a wide class of problems and can be generalized to handle problems defined over domains that are not cartesian product of intervals. The transformation technique *reshape* has been used three times in this paper and it seems very mechanical. In fact, a metalanguage has been designed and implemented to facilitate the mechanical aspect of the program transformation[12]. Reshape has been implemented as a meta procedure to derive new program automatically when given an old program and a pair of bijective mappings.

Though this paper stops at an implementation targeted for a mesh of processors, implementations for hypercube can be derived similarly by using another pair of mappings to efficiently embed a two-dimensional mesh into a hypercube [6]. Efficient implementations for shared memory MIMD machines can be also derived naturally by viewing shared memory processors as a fully connected network of processors. Therefore a two-dimensional mesh can be embedded easily.

References

[1] A. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1981.

[2] Marina Chen, Young-il Choo, and Jingke Li. Compiling parallel programs by optimizing performance. *The Journal of Supercomputing*, 1(2):171–207, July 1988.

[3] Marina C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, pages 461–491, December 1986.

[4] A. Church. *The Calculi of Lambda Conversion.* Princeton University Press, Princeton, NJ, 1941.

[5] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lysenga, Steve W. Otto, John K. Salmon, and David W. Walker. *Solving Problems on Concurrent Processors*. Prentice Hall, 1988.

[6] Ching-Tien Ho and S. Lennart Johnsson. Embedding meshes in Boolean cubes by graph decomposition. *Journal of Parallel and Distributed Computing*, 8(4):325–339, April 1990.

[7] Michel Jacquemin and J. Allan Yang. Crystal reference manual, version 3.0. Technical Report YALEU/DCS/TR-840, Dept. of Computer Science, Yale University, January 1991.

[8] Jingke Li and Marina Chen. Generating explicit communication from shared-memory program references. In *Supercomputing 90*, New York, NY, Nov. 1990.

[9] Jingke Li and Marina Chen. Index domain alignment: Minimizing cost of cross-reference between distributed arrays. In *Proceedings of the 3rd Symposium on the Frontiers of Massively Computation*, College Park, Maryland, Oct. 1990.

[10] Jingke Li and Marina Chen. Automating the coordination of interprocessor communication. In D. Gerlernter et al., editor, *Programming Languages and Compilers for Parallel Computing*. The MIT Press, 1991.

[11] R. Milner. A proposal for standard ML. In *ACM Symp. on LISP and Functional Programming*, pages 184–197, Aug. 1984.

[12] J. Allan Yang and Young-il Choo. Parallel-program transformation using a metalanguage. In *Proceedings of The Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Williamsburg, Virginia*, pages 11–20, April 1991.

Appendix A Brief Notes on the Lambda Notation

Because the lambda notation is used in Crystal for representing function and some properties of the lambda calculus [4] are used for transforming programs, we provide a very brief introduction to the the lambda notation. The lambda notation gives us a way to express a function without giving an explicit name. For example, $\lambda x.x + 1$ is an anonymous function that takes an argument x (i.e., argument is written after λ) and returns a value $x + 1$ (i.e., the function body is written after the "." and extends to the right as far as possible). If we want to give $\lambda x.x + 1$ a name f, we write it as $f = \lambda x.x + 1$. The traditional way is to write it as $f(x) = x + 1$. We write $(\lambda x.x + 1)(3)$ to mean $\lambda x.x + 1$ is applied to 3. That is, we simply concatenate the function and the argument. We call an expression like $\lambda x.x + 1$, a *λ-abstraction* term, and expressions like $(\lambda x.x + 1)(3)$ or $f(3)$, *application* terms. A *β-redex* is an application term with the rator (i.e., the function) being a λ-abstraction term, e.g., $(\lambda x.x + 1)(y)$ is a β-redex. To *β-reduce* a β-redex is to carry out the application of that redex, e.g., β-reducing $(\lambda x.x + 1)(y)$ will result in $y + 1$. Let f be any function, doing *η-abstraction* on f results in an equivalent function $\lambda x.(f(x))$. That is, f is equivalent to $\lambda x.(f(x))$ by η-abstraction.