THE DERIVATION OF SYSTOLIC IMPLEMENTATIONS OF PROGRAMS

Chua-Huang Huang and Christian Lengauer

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-86-10 April 1986
Revised, January 1987

Abstract

We present a mathematically rigorous and, at the same time, convenient method for systolic design and derive systolic designs for two matrix computation problems. Each design is synthesized from a simple program and a proposed layout of processors. The synthesis derives a systolic parallel execution, channel connections for the proposed processor layout, and an arrangement of data streams such that the systolic execution can begin. Our choices of designs are governed by formal theorems. The synthesis method is implementable and is particularly effective if implemented with graphics capability. Our implementation on the Symbolics 3600 displays the resulting designs and simulated executions graphically on the screen. The method's centerpiece, a transformation of sequential program computations into systolic parallel ones, has been mechanically proved correct.

This research was partially supported by Grant No. 26-7603-35 from the Lockheed Missiles & Space Corporation and by Grant No. DCR-8610427 of the National Science Foundation.

Table of Contents

1. Introduction

The development of programs need not immediately address implementation concerns. Instead, one can proceed in stages.

(1) One can first derive a *program*. A program is an abstract solution that conforms with the problem specification and is implementable in some way. The purpose of a program is not to address every implementation issue. When developing programs, we are concerned with issues of correctness.

(2) Then one can derive an *execution* or a *trace*. A trace represents a more concrete solution than a program. The purpose of a trace is to resolve issues of execution that the program leaves open. When developing traces, we are concerned with issues of efficiency.

(3) Finally, one might outline a *computer architecture*. An architecture is a partial description of some hardware configuration. In developing architectures, we are, again, concerned with issues of efficiency. In addition, we have to settle a question of consistency: our hardware solution, the architecture, must be able to execute our software solution, the trace.

This division of concerns can be of great help in program development. In the best of all worlds, where there is a proven, mechanical way to obtain efficient, complicated executions from simple programs, the programmer derives his understanding of the software solution mostly from the program and is hardly ever concerned with executions. In fact, he or she might even get help in constructing a suitable computer architecture without delving into the intricacies of program execution. Our work aims at such a world. Currently, we are restricting ourselves to quite specific programming problems: in our previous work sorting problems and in this paper matrix computation problems.

The concept of execution that we are concentrating on is sequencing. Our programs leave the question of sequencing open; they may result in simple, i.e., sequential or in complicated, i.e., parallel executions. Our derivation of executions from programs is mechanical. A program is first translated into a sequential execution which is then transformed into a shorter parallel execution, if possible. All our trace transformations follow the same formally defined transformation strategy.

We will present two matrix computation programs: matrix multiplication and LU-decomposition, and will automatically derive parallel executions for them. We will then proceed to propose systolic designs that can perform these parallel executions. A *systolic design* is a network of processors that are connected in simple patterns and perform simple operations under global synchronization [10]. The description of a systolic design will comprise a processor layout, channel connections between the processors, and the layout and direction of data travelling through the network. We will only have to propose the layout of the processors. If the layout is suitable for our execution, the rest can be synthesized automatically. After scrutiny of the resulting design, we may want to improve it by altering either the processor layout or the program. In one of our examples, matrix multiplication, we will make one adjustment to the processor layout and one adjustment to the program. Our search for alternative designs is guided by a number of theorems about our design method.

2. The Method: Programs, Traces, and Trace Transformations

2.1. Programs

Programs are expressed in a refinement language with the following features:

- The definition of a *refinement* consists of a refinement name with an optional list of formal parameters, separated by a colon from a refinement body. An entry condition involving the formal parameters may be added in curly brackets. (See an example below.) The following are the only three choices of refinement body.

- The *null statement*, skip, does nothing.

- The *basic statement* is a statement that is not refined any further. It is denoted by a name and a list of parameters. The basic statement is of an imperative nature, i.e., its implementation requires updates.

- The *composition* *S0*;*S1* of refinements *S0* and *S1* applies *S1* to the results of *S0*. Each of *S0* and *S1* can be a refinement call (i.e., a refinement name, maybe, with an actual parameter list), a basic statement, or the null statement. Sequences of compositions *S0*;*S1*;...;*Sn* are also permitted. Refinement calls may be recursive.

For example, a program that applies some basic statement, stat, n times is:

$\{n{=}0\}$ $S(n)$: skip
$\{n{>}0\}$ $S(n)$: stat; $S(n{-}1)$

Several definitions of the same refinement, usually prefixed with different entry conditions, may be provided. They represent a case analysis. We have here provided two definitions of refinement S. As a short-hand, we shall omit an entry condition that requires a specific value for a parameter and shall, instead, specify that value at the appropriate place in the parameter list:

$S(0)$: skip
$\{n{>}0\}$ $S(n)$: stat; $S(n{-}1)$

There may be a variety of basic statements. The language of sorting networks has one basic statement: the comparator module [9]. The most frequently used basic statement of the language of matrix computations is the inner product step [10].

2.2. Traces

Following the conventional implementation of composition as sequential execution, a sequential execution is obtained from a refinement by replacing every semicolon with a right-pointing arrow. That is, program *S0*;*S1* has trace $S0 \rightarrow S1$.[1] This implementation of composition is always safe, but it may be overly restrictive. We can transform it into different executions with the same effect. Such transformations can relax sequencing and incorporate parallelism into executions. We denote parallel execution by angle brack-

[1]We also eliminate all null statements. In other words, program skip has the empty trace.

ets. Thus, we will, in certain cases, execute program *S0;S1* by trace *<S0 S1>*. We call *<S0 S1>* a *parallel command*.

We denote the length of a trace S by $|S|$ and define it as follows:

$|\text{stat}| =_{\text{def}} 1$ for every basic statement stat

$|S0 \to S1| =_{\text{def}} |S0| + |S1|$

$|<S0\ S1>| =_{\text{def}} \max(|S0|,|S1|)$

The length of a trace serves as an estimate of the trace's execution time. Our estimate is rather crude. For more accurate estimates, the previous definitions can be adjusted accordingly.

We want to define a transformation strategy formally as a function that operates on a data structure. Therefore, we need to be more specific about the representation of traces. We represent a trace by a list. The atoms of the list are the basic statements in the trace. Alternating list levels represent sequential and parallel execution, respectively. The top level represents sequential execution. Our notation follows LISP. For example, trace *S0→S1* is represented as (*S0 S1*), and trace *<S0 S1>* as ((*S0 S1*)). *Nil* denotes the empty list. We also adopt the LISP list operations: *cons*, *car*, and *cdr*.

2.3. Trace Transformations

Our intent is to shorten the length of a trace by a sequence of transformations. Each transformation must preserve the trace's effect. Trace transformations are justified by semantic relations that program components may or may not satisfy:

(1) A program component S that is *idempotent* can be executed once or any number of times consecutively with identical effect. Thus, *S→S* in a trace may be transformed to S, and vice versa. The idempotence of S is declared as: idem S.

(2) A program component S that is *neutral* has no effect other than that it may take time to execute. Thus, S may be omitted from or added to a trace. Neutrality implies idempotence. The neutrality of S is declared as: ntr S.

(3) Two program components *S0* and *S1* that are *commutative* can be executed in any order with identical effect. Thus, *S0→S1* in a trace may be transformed to *S1→S0*. The commutativity of *S0* and *S1* is declared as: *S0* com *S1*.

(4) Two program components *S0* and *S1* that are *independent* can be executed in parallel and in sequence with identical effect. Thus *S0→S1* in a trace may be transformed to *<S0 S1>*. Independence implies commutativity. The independence of *S0* and *S1* is declared as: *S0* ind *S1*.

Semantic relations are made explicit by declarations that accompany the refinement program. The format of a semantic declaration is:

enabling predicate ⇒ *semantic relation*

The enabling predicate is a condition on the parameters of the program components that are semantically related. Just like the correctness of refinements, the correctness of semantic declarations can be proved formally [15].

Since transformations exploit semantic relations, we need recognizer functions for semantic relations if we want to define a transformation strategy. We postulate the existence of functions *idem*(*S*), *ntr*(*S*), *com*(*S0*,*S1*), and *ind*(*S0*,*S1*) that recognize the idempotence, neutrality, commutativity, and independence of basic statements. We shall apply these functions to the atoms of our list representation of traces.

We shall also need to establish the commutativity and independence of some atom i with a list l of atoms. We use two functions:

```
is-com(i,l)  =_def  if l=nil
                      then true
                      else com(i,car(l)) ∧ is-com(i,cdr(l))

is-ind(i,l)  =_def  if l=nil
                      then true
                      else ind(i,car(l)) ∧ is-ind(i,cdr(l))
```

3. The Transformation Strategy

Transform is a function that maps traces to traces. *Transform* applies to sequential traces only. We have the traces in mind that are trivially derived from refinements. *Transform* ravels the basic statements of the sequential trace, one by one, starting with the right-most element and discarding neutral basic statements. The result of *transform* is a trace containing a sequence of parallel commands (some of which may only contain a single basic statement). Given a sequential trace l in list representation, the transformation strategy, *transform*, is formally defined as follows:

```
transform(l)  =_def  remove-all-ntr(ravel-trans(l))
```

Function *ravel-trans* represents the compression of the sequential trace into a parallel trace; function *remove-all-ntr* represents the elimination of neutral trace atoms. Since *ravel-trans* provides the input to *remove-all-ntr*, let us first define *ravel-trans*:

```
ravel-trans(l)  =_def  if l=nil
                         then nil
                         else ravel(car(l),ravel-trans(cdr(l)))
```

Ravel-trans recursively combines applications of function *ravel*. *Ravel* accepts a trace atom i and a parallel trace l, and returns i ravelled into l. It is defined as:

```
ravel(i,l)  =_def  if idem(i) ∧ idem-applies(i,l)
          (1)        then l
                     else if ind-applies(i,l)
          (2)               then merge(i,l)
          (3)               else commute(i,l)
```

Trace atom i is discarded if it is idempotent and there exists a parallel command of l in which i occurs and all trace atoms prior to which are commutative with i (case 1). The latter condition is recognized by function *idem-applies*(i,l). Otherwise, function *ind-applies*(i,l) establishes whether i is independent with a parallel command of l. If so, it is merged with the right-most parallel command which is independent of i and all trace atoms prior to which are commutative with i (case 2). If not, i is commuted as far as possible to the right in l and forms a single-atom parallel command (case 3).

The formal definition of function *idem-applies* is:

```
idem-applies(i,l) =_def if l=nil
                            then false
                            else if i∈car(l)
                                    then true
                                    else is-com(i,car(l)) ∧ idem-applies(i,cdr(l))
```

The formal definition of function *ind-applies* is:

```
ind-applies(i,l) =_def if l=nil
                           then false
                           else if is-ind(i,car(l))
                                   then true
                                   else is-com(i,car(l)) ∧ ind-applies(i,cdr(l))
```

The function that merges i with a parallel command of l is:

```
merge(i,l) =_def if l=nil
                     then cons(cons(i,nil),nil)
                     else if ind-applies(i,cdr(l))
                             then cons(car(l),merge(i,cdr(l)))
                             else cons(cons(i,car(l)),cdr(l))
```

The value of expression *cons*(*cons*$(i,nil),nil)$ is $((i))$. The value of expression *cons*(*cons*$(i,car(l)),cdr(l))$ is trace l with atom i added to the first parallel command.

The function that commutes i with a parallel command of l is:

```
commute(i,l) =_def if l=nil
                       then cons(cons(i,nil),nil)
                       else if is-com(i,car(l))
                               then cons(car(l),commute(i,cdr(l)))
                               else cons(cons(i,nil),l)
```

The value of expression *cons*(*cons*$(i,nil),l)$ is trace l with the single-atom parallel command (i) added to the front.

This concludes the definition of *ravel-trans*. The transformation strategy for sorting networks described in [16] - let us call it here *transform-sort* - is *ravel-trans* with a simplified ravelling function, *ravel-sort*, in place of *ravel*:

```
$ravel\text{-}sort(i,l)$ $=_{\text{def}}$ if $idem(i) \wedge idem\text{-}applies(i,l)$
                          then $l$
                          else if $ind\text{-}applies(i,l)$
                                  then $merge(i,l)$
                                  else $cons(cons(i,nil),l)$
```

Let us now move on to the treatment of neutrality. We define function *remove-all-ntr* as:

```
$remove\text{-}all\text{-}ntr(l)$ $=_{\text{def}}$ if $l{=}nil$
                             then $nil$
                             else $cons(remove\text{-}ntr(car(l)),remove\text{-}all\text{-}ntr(cdr(l)))$
```

Remove-all-ntr applies *remove-ntr* recursively to every parallel command in the argument trace. *Remove-ntr* accepts a parallel command and returns that command without neutral trace atoms:

```
$remove\text{-}ntr(l)$ $=_{\text{def}}$ if $l{=}nil$
                        then $nil$
                        else if $ntr(car(l))$
                                then $remove\text{-}ntr(cdr(l))$
                                else $cons(car(l),remove\text{-}ntr(cdr(l)))$
```

This concludes the definition of *transform*. Just as for *transform-sort*, we have checked mechanically with the Boyer-Moore theorem prover [1] that *transform* preserves the semantics of its argument trace, and that its result trace does not contain any parallel command with dependent trace atoms, i.e., with incorrect parallelism.

The basic statement of sorting networks is the comparator module. The comparator module compares two elements of the sequence to be sorted and, if necessary, exchanges them into order. Comparator modules are not neutral. Also distinct dependent comparator modules are not commutative. Neutrality and commutativity that is not implied by independence are the only two properties that *transform* exploits but *transform-sort* does not exploit. Therefore, if applied to sorting networks, the result of *transform* is that of *transform-sort*: both represent the same optimal transformation strategy. (We have proved that mechanically, too.) For other programming languages in which two distinct dependent basic statements may be commutative, like matrix computations, the transformation strategy may not identify the commutation that yields the shortest execution. For example, in our matrix multiplication example, we will have to search for that particular commutation.

4. Matrix Computations

4.1. Programs

The central basic statement[2] of the language of *matrix computations* is the *inner product step*, written ips(a,b,c). It involves three distinct variables a, b, and c and performs the following assignment:

$$c := c + a * b$$

We consider matrices whose non-zero values are concentrated in a "band" around the diagonal. An inner product step ips(a,b,c) containing off-band elements a or b does not change the value of c, i.e., is neutral. We exploit this neutrality. To identify off-band elements of the matrix, we must precisely describe the width of the band of non-zero elements around the diagonal. This *band width* is determined by two natural numbers: the largest distance p, of a potentially non-zero element in the upper triangle from the diagonal, and the largest distance, q, of a potentially non-zero element in the lower triangle from the diagonal.[3]

Only neutral inner product steps are idempotent. Since we exploit their neutrality, we do not exploit their idempotence.

On a parallel architecture that permits the sharing of variables, two inner product steps ips($a0$,$b0$,$c0$) and ips($a1$,$b1$,$c1$) are independent if their target variables $c0$ and $c1$ are distinct.[4] But we are interested in executions on particular, systolic architectures that do not permit the sharing of variables. Therefore, we must use a stronger independence criterion and require that $a0$ and $a1$ are distinct, $b0$ and $b1$ are distinct, and $c0$ and $c1$ are distinct. Recall that the three variables of an individual inner product step are distinct by assumption.

All inner product steps are commutative. This makes commutativity, per se, meaningless. We do not exploit commutativity in trace transformations unless it is a consequence of independence.

4.2. Systolic Designs

A parallel trace specifies a partial order of basic statements without reference to a particular architecture. We will develop systolic arrays that can execute the parallel trace. We specify a systolic array with the help of four functions.

[2] We shall define additional basic statements where necessary.

[3] The *distance* of a matrix element from the diagonal is the absolute value of the difference of its two indices.

[4] See the Independence Theorem of [14].

The first two functions are called *step* and *place*. The domain of both functions is the set of basic statements that occur in the parallel trace. *Step* determines when basic statements are to be executed, and *place* determines where basic statements are to be executed.[5]

Step maps basic statements to the integers. The intention is to count the parallel commands of the parallel trace in their order of execution. *Step* is derived from the parallel trace. The derivation of *step* must adhere to two conditions:

(S1) basic statements of the same parallel command must be mapped to the same integer,

(S2) basic statements of adjacent parallel commands must be mapped to consecutive integers.

We are free to choose an appropriate integer, *fs*, for the basic statements of the first parallel command. If *step* satisfies conditions (S1) and (S2), any two basic statements in the same parallel command must have identical step values. *Step* can be derived by solving a system of equations whose formulation is guided by conditions (S1) and (S2) (see the next section).

Place maps basic statements to an integer space of some dimension d. We assume that every point of that space is occupied by a processor. The intention is to assign basic statements to the processors. Processors that are not assigned a statement at some step simply forward the data on their input channels to their output channels during that step. Processors that are at no step assigned a statement need not be implemented. *Place* is not derived from the parallel trace but proposed separately. *Place* has to satisfy the following condition:

(P1) basic statements of the same parallel command must be assigned distinct points.

We have a simple condition that establishes whether our proposals for *place* satisfy (P1) (see the next section).

In programs, data are represented by variables. In systolic computations, data, i.e., variables travel between processors. A variable may be accessed by one processor at one step and by another processor at a later step. We have to specify a layout and flow of variables that provides each processor with the expected inputs at the steps at which it is supposed to execute its basic statement. At present, our method is confined to systolic arrays in which processors are only connected by unidirectional channels to processors that occupy neighboring points.[6] For designs with these characteristics, we can synthesize the input pattern and flow of data from *step* and *place*. To this end, we introduce two more functions: *pattern* and *flow*. The domain of both functions is the set of program variables. *Flow* specifies the direction of data movement, and *pattern* specifies the initial data layout.

[5]In general, we must distinguish multiple occurrences of identical basic statements - by some sort of counter, say. However, we omit this trivial complication here. Our programming examples lead to traces whose basic statements are all distinct.

[6]Two points $(p_0,\dots,p_{d-1})$ and $(q_0,\dots,q_{d-1})$ of the d-dimensional integer space are *neighbors* if $0 \le |p_i - q_i| \le 1$, where $0 \le i < d$.

Flow maps program variables to the same d-dimensional integer space as *place*. The intention is to indicate, for every processor in the network, which of its neighbors receive its output values at the next execution step, i.e., to which of its neighbors it must be connected by an outgoing channel. *Flow* is synthesized from *step* and *place* as follows: if variable v is accessed by distinct basic statements $s0$ and $s1$,

$$flow(v) \;=_{\text{def}}\; (place(s1)-place(s0))/(step(s1)-step(s0))$$

For variables v that are accessed by only one basic statement, we must provide the definition of *flow* explicitly. *Flow* is only well-defined if its images do not depend on the particular choice of pairs $s0$ and $s1$.

Pattern maps program variables to the same space as *place*. The intention is to lay out the input data for the various processors in an initial pattern such that the systolic execution can begin. (*Flow* describes the propagation of the data towards and through the network as the execution proceeds.) With constant fs being the arbitrary step value that we choose for the first parallel command, *pattern* is synthesized from *step*, *place*, and *flow* as follows: if variable v is accessed by basic statement s,

$$pattern(v) \;=_{\text{def}}\; place(s)-(step(s)-fs)*flow(v)$$

Pattern is only well-defined if its images do not depend on the particular choice of basic statement s. With *pattern* specifying the initial data layout, we can derive the data layout for successive steps of the systolic execution: the data layout after k steps is given by $pattern(v)+k*flow(v)$.

4.3. The Graphics System

We have implemented the transformation strategy and the computation of the previous functions in a graphics system on the Symbolics 3600. Our system can display two-dimensional processor layouts and simulate sequences of execution steps on them. At any fixed step, it displays the data layout and flow and indicates the active processors.

The three central commands of the graphics system are *add-processor*, *add-design*, and *display-design*. *Add-processor* adds the specification of a processing element, which consists of the name of the basic statement the processor is supposed to apply, its number of arguments, and the identifiers of its input and output variables. *Add-design* asks for a design name and the four components that are necessary to synthesize a design: a program refinement, semantic declarations, a step, and a place function. At present, we require the explicit specification of *step*, even though it could be synthesized from the parallel trace (see the following section). *Display-design* takes a design name and a refinement call. It displays the data layout for the submitted call on the submitted processor layout and simulates the systolic execution. The simulation can be controlled to advance or to back up a number of steps. At each step, the active processors are highlighted. The simulation of a systolic execution is particularly helpful for systolic arrays with complex data flow [8].

The figures in this paper are hard-copies of images produced by our system.

5. Theorems for Linear Systolic Designs

A number of researchers have analyzed systolic designs with notions of linear algebra [17, 18, 20, 21]. We shall do something similar here. In this section, we investigate a specific class of systolic designs: linear systolic designs. We defer the proofs of theorems to the appendix.

A systolic design is *linear* if it is specified by linear step and place functions. Linear systolic designs are particularly interesting because their data movement proceeds at a fixed rate in straight lines. We limit our theoretical discussion to programs with only one type of basic statement.[7] Let us denote the basic statement by $s(x_0,x_1,\ldots,x_{r-1})$. Also, we use $s[x_i'/x_i]$ to denote the substitution of x_i' for argument x_i in basic statement $s(x_0,x_1,\ldots,x_{r-1})$.

Formally, a systolic design is linear, if *step* and *place* are described by the following linear equations:

(E1) $\mathit{step}(s(x_0,x_1,\ldots,x_{r-1})) = \alpha_{0,0}x_0+\alpha_{0,1}x_1+\ldots+\alpha_{0,r-1}x_{r-1}+\alpha_{0,r}$

(E2) $\mathit{place}(s(x_0,x_1,\ldots,x_{r-1})) = (\alpha_{1,0}x_0+\alpha_{1,1}x_1+\ldots+\alpha_{1,r-1}x_{r-1}+\alpha_{1,r},\ \ldots,\ \alpha_{d,0}x_0+\alpha_{d,1}x_1+\ldots+\alpha_{d,r-1}x_{r-1}+\alpha_{d,r})$

where the range of *place* is the d-dimensional integer space. In a non-linear systolic design, equations (E1) and (E2) would be of a higher degree. We shall explain the derivation of *step* and discuss theorems about *place*, *flow*, and *pattern* that provide guidance in the choice of a place function.

Consider a non-empty parallel trace. The images of its individual basic statements under *step*, as defined in (E1), constitute a set of linear formulas. Take the image of the first basic statement in the parallel trace and equate it with a chosen number. Impose conditions (S1) and (S2) to derive equations for the other basic statements. The result is a set of linear equations in the variables $\alpha_{0,0}$, $\alpha_{0,1}$, ..., $\alpha_{0,r-1}$, and $\alpha_{0,r}$, whose solution determines *step*. However, the equations do not guarantee the existence of a unique solution. For example, if the parallel trace consists of only one statement, there are infinitely many solutions for *step*, all of which satisfy conditions (S1) and (S2). It is also possible that no solution exists at all.

While conditions (S1) and (S2) are, generally, sufficient to synthesize *step*, condition (P1) is not sufficient to synthesize *place*. We must propose *place* independently and test whether it satisfies (P1). The following theorem provides such a test.

Theorem 1: Let *step* be a linear step function for parallel trace t that satisfies (S1) and (S2). Let *place* be a linear place function for t. *Place* satisfies (P1) if the following equations have the zero vector as the unique solution:

[7] This restriction is not as severe as it may seem. While matrix multiplication only requires single-type basic statements, we will be able to apply our theorems also to LU-decomposition, which uses basic statements of several types.

$$\alpha_{0,0}u_0+\alpha_{0,1}u_1+\ldots+\alpha_{0,r-1}u_{r-1}=0$$
$$\alpha_{1,0}u_0+\alpha_{1,1}u_1+\ldots+\alpha_{1,r-1}u_{r-1}=0$$
$$\ldots$$
$$\alpha_{d,0}u_0+\alpha_{d,1}u_1+\ldots+\alpha_{d,r-1}u_{r-1}=0$$

where r is the number of arguments of basic statement s. In particular, if *place* maps to $r-1$ dimensions, i.e., $d=r-1$, *place* satisfies (P1) if the coefficient determinant of this previous system of equations is not zero.

Given a linear step function satisfying (S1) and (S2) and a linear place function satisfying (P1), we can compute *flow* and *pattern*. The computation of *flow* and *pattern* must be well-defined, that is, their result must not depend on the choice of basic statements. Matrix computation programs use subscripted variables. In our programming language, the variable subscripts appear as arguments of the program's basic statements. If the variable subscripts are determined by $r-1$ arguments of the r-argument statement, then the flow of the variable derived from *step* and *place* is well-defined. This property is stated in Theorem 2. In our programming example, matrix multiplication, matrix elements accessed by a basic statement are determined each by two of the statement's three arguments (Section 4).

Theorem 2: Let *step* be a linear step function for parallel trace t that satisfies (S1) and (S2). Let *place* be a linear place function for t that satisfies (P1). If the subscripts of variable v are determined by all but one of the r arguments of the basic statement, then *flow* is well-defined for variable v.

Given a parallel trace t which satisfies conditions (S1), (S2), and (P1), no two basic statements in t can be identical. If a variable's subscripts are determined by all r, not just $r-1$, arguments of a basic statement, this variable can be accessed by at most one basic statement. Therefore, we cannot derive its flow function but have to provide it explicitly. In general, while the processor layout for a program with r-argument basic statements requires dimension $r-1$, the data layout requires dimension r. An example is matrix-vector multiplication [10].

Given a step function satisfying (S1) and (S2), a place function satisfying (P1), and a well-defined flow function, the derived pattern function is well-defined. This property is stated by Theorem 3.

Theorem 3: Let *step* be a linear step function for parallel trace t that satisfies (S1) and (S2). Let *place* be a linear place function for t that satisfies (P1). Let *flow*, derived from *step* and *place*, be well-defined. Then *pattern*, derived from *step*, *place*, and *flow*, is well-defined.

6. Examples

The following subsections describe two matrix computation problems: matrix multiplication and LU-decomposition. The treatment of both examples follows the same path: the problem is stated, a refinement program that is a solution is presented, semantic relations are added, and a sequential trace is derived from the refinement. Then, *transform* is applied to exploit the semantic relations, a parallel trace is obtained, and a systolic design that can execute the parallel trace is defined. For matrix multiplication, we shall derive three alternative designs. The systolic implementations we present are well-known. Our point is that they can be obtained as result of a synthesis method that considers matters of execution after the derivation of a correct program.

6.1. Matrix Multiplication

The problem is to multiply two distinct $n \times n$ matrices A and B and assign the product to a third $n \times n$ matrix C, such that

$$c_{i,j} = \sum_{k=0}^{n-1} (a_{i,k} * b_{k,j}) \qquad \text{for} \quad 0 \leq i \leq n-1 \text{ and } 0 \leq j \leq n-1$$

In the solution to this problem, the inner product step takes the form ips$(a_{i,k}, b_{k,j}, c_{i,j})$. If we fix variables A, B, and C, we can express the inner product step solely in terms of the matrix indices i, j, and k. We shall use the notation $(i{:}j{:}k)$.

With inner product steps, the following program is the standard solution to matrix multiplication; it is assumed that matrix C is initially everywhere zero:

```
for i from 0 to n−1 do
    for j from 0 to n−1 do
        for k from 0 to n−1 do (i:j:k)
```

Translated to our programming language, this program becomes:

```
         matrix-matrix(n):          product(n−1,n−1)

         product(0,m):              row(0,m,m)
{0<i}    product(i,m):              product(i−1,m); row(i,m,m)

         row(i,0,m):                inner-product(i,0,m)
{0<j}    row(i,j,m):                row(i,j−1,m); inner-product(i,j,m)

         inner-product(i,j,0):      (i:j:0)
{0<k}    inner-product(i,j,k):      inner-product(i,j,k−1); (i:j:k)
```

This rather complex syntax has the advantage that each composition of two basic statements is represented explicitly by a semicolon. This will simplify the translation of the program into a sequential execution.

The declarations of neutrality and independence for inner product step $(i{:}j{:}k)$ are:

(1) $p_A<k-i \vee q_A<i-k \vee p_B<j-k \vee q_B<k-j \quad \Rightarrow \quad \mathrm{ntr}\ (i{:}j{:}k)$

(2) $(i_0\neq i_1 \vee j_0\neq j_1) \wedge (i_0\neq i_1 \vee k_0\neq k_1) \wedge (j_0\neq j_1 \vee k_0\neq k_1) \quad \Rightarrow \quad (i_0{:}j_0{:}k_0)\ \mathrm{ind}\ (i_1{:}j_1{:}k_1)$

where p_A and q_A are the band widths of the upper and the lower triangle of matrix A, and p_B and q_B are the band widths of the upper and the lower triangle of matrix B. The upper and the lower triangle of result matrix C have band widths $p_C=p_A+p_B$ and $q_C=q_A+q_B$. The enabling predicate of the neutrality declaration identifies inner product steps that access off-band elements of matrices A or B; the enabling predicate of the independence declaration identifies inner product steps that do not access common variables.

To demonstrate the incremental nature of our method, we develop three different designs for matrix multiplication.

6.1.1. The First Design

We obtain a sequential trace by replacing ";" with "→". For example, the sequential trace for the multiplication of two 4×4 matrices (*matrix-matrix*(4)) expands to:

```
(0:0:0)→(0:0:1)→(0:0:2)→(0:0:3)→
         (0:1:0)→(0:1:1)→(0:1:2)→(0:1:3)→
                  (0:2:0)→(0:2:1)→(0:2:2)→(0:2:3)→
                           (0:3:0)→(0:3:1)→(0:3:2)→(0:3:3)→
         (1:0:0)→(1:0:1)→(1:0:2)→(1:0:3)→
                  (1:1:0)→(1:1:1)→(1:1:2)→(1:1:3)→
                           (1:2:0)→(1:2:1)→(1:2:2)→(1:2:3)→
                                    (1:3:0)→(1:3:1)→(1:3:2)→(1:3:3)→
                  (2:0:0)→(2:0:1)→(2:0:2)→(2:0:3)→
                           (2:1:0)→(2:1:1)→(2:1:2)→(2:1:3)→
                                    (2:2:0)→(2:2:1)→(2:2:2)→(2:2:3)→
                                             (2:3:0)→(2:3:1)→(2:3:2)→(2:3:3)→
                           (3:0:0)→(3:0:1)→(3:0:2)→(3:0:3)→
                                    (3:1:0)→(3:1:1)→(3:1:2)→(3:1:3)→
                                             (3:2:0)→(3:2:1)→(3:2:2)→(3:2:3)→
                                                      (3:3:0)→(3:3:1)→(3:3:2)→(3:3:3)
```

The parallel trace is obtained by letting *transform* exploit the previous declarations of neutrality and independence on this sequential trace. The ten commands of the parallel trace with neutral atoms collect the inner process steps of the previous sequential trace by columns:

```
  <(0:0:0)>
→ <(0:0:1) (0:1:0) (1:0:0)>
→ <(0:0:2) (0:1:1) (0:2:0) (1:0:1) (1:1:0) (2:0:0)>
→ <(0:0:3) (0:1:2) (0:2:1) (0:3:0) (1:0:2) (1:1:1) (1:2:0) (2:0:1) (2:1:0) (3:0:0)>
→ <(0:1:3) (0:2:2) (0:3:1) (1:0:3) (1:1:2) (1:2:1)
                                  (1:3:0) (2:0:2) (2:1:1) (2:2:0) (3:0:1) (3:1:0)>
→ <(0:2:3) (0:3:2) (1:1:3) (1:2:2) (1:3:1) (2:0:3)
                                  (2:1:2) (2:2:1) (2:3:0) (3:0:2) (3:1:1) (3:2:0)>
→ <(0:3:3) (1:2:3) (1:3:2) (2:1:3) (2:2:2) (2:3:1) (3:0:3) (3:1:2) (3:2:1) (3:3:0)>
→ <(1:3:3) (2:2:3) (2:3:2) (3:1:3) (3:2:2) (3:3:1)>
→ <(2:3:3) (3:2:3) (3:3:2)>
→ <(3:3:3)>
```

As band width, let us assume $p_A=q_A=p_B=q_B=1$. Elimination of all neutral atoms yields the target trace:

```
  <(0:0:0)>
→ <(0:0:1) (0:1:0) (1:0:0)>
→ <(0:1:1) (1:0:1) (1:1:0)>
→ <(0:2:1) (1:1:1) (2:0:1)>
→ <(1:1:2) (1:2:1) (2:1:1)>
→ <(1:2:2) (2:1:2) (2:2:1)>
→ <(1:3:2) (2:2:2) (3:1:2)>
→ <(2:2:3) (2:3:2) (3:2:2)>
→ <(2:3:3) (3:2:3) (3:3:2)>
→ <(3:3:3)>
```

This trace has length 10. For $n\times n$ input matrices, the length of the parallel trace is $3n-2$. The length of the trace is independent of the band width, but the band width influences the width of the trace, i.e., the degree of concurrency.

The step function is derived from the parallel trace. Let the step function be a linear function:

$$step((i{:}j{:}k)) \;=\; \alpha_0*i+\alpha_1*j+\alpha_2*k+\alpha_3$$

Recall that we are allowed to choose the step value of the first parallel command. We choose the value to make the constant term, α_3, 0. In this case, the step value of the first parallel command is 0. Applying the step function to the basic statements in the first two parallel commands of the above parallel trace, we obtain the following equations:

$$\begin{aligned} step((0{:}0{:}0)) &= \alpha_3 &= 0 \\ step((0{:}0{:}1)) &= \alpha_2+\alpha_3 &= 1 \\ step((0{:}1{:}0)) &= \alpha_1+\alpha_3 &= 1 \\ step((1{:}0{:}0)) &= \alpha_0+\alpha_3 &= 1 \end{aligned}$$

The solution to these equations is $\alpha_0=\alpha_1=\alpha_2=1$ and $\alpha_3=0$. The solution is consistent for the equations obtained by applying the step function to the rest of the basic statements. Therefore, the derived step function is:

$$step((i{:}j{:}k)) \;=\; i+j+k$$

The place function cannot be derived from the parallel trace but must be proposed separately. It seems promising to lay the processors out in a plane, i.e., in our method on the two-dimensional integer lattice. Our first idea is to assign each basic statement to the point whose coordinates match the indices of the statement's target variable. This decision is rather arbitrary. At this stage, we do not have any information that might guide us in the choice of a processor layout. As we shall see later, other layouts are possible. Inner product step $(i{:}j{:}k)$ has target variable $c_{i,j}$. We propose:

$$place((i{:}j{:}k)) =_{\text{def}} (i,j)$$

The dimension of *place* is two which is one less than the number of the arguments in $(i{:}j{:}k)$. By Theorem 1, *place* satisfies condition (P1), because the determinant constructed from the coefficients of *step* and *place* is not zero:

$$\begin{vmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{vmatrix} = 1$$

where the first row, (1 1 1), is constructed from *step*, the second row, (1 0 0), from the first dimension of *place*, and the third row, (0 1 0), from the second dimension of *place*.

Variable $a_{i,k}$ appears in basic statements $(i{:}j{:}k)$ and $(i{:}j{+}1{:}k)$, and these two statements are executed in consecutive steps. Therefore, we can derive the flow of $a_{i,k}$:

$$\begin{aligned} flow(a_{i,k}) &= (place((i{:}j{+}1{:}k)) - place((i{:}j{:}k)))/(step((i{:}j{+}1{:}k)) - step((i{:}j{:}k))) \\ &= (0,1) \end{aligned}$$

Similarly, we derive the flows of $b_{k,j}$ and $c_{i,j}$:

$$\begin{aligned} flow(b_{k,j}) &= (place((i{+}1{:}j{:}k)) - place((i{:}j{:}k)))/(step((i{+}1{:}j{:}k)) - step((i{:}j{:}k))) \\ &= (1,0) \end{aligned}$$

$$\begin{aligned} flow(c_{i,j}) &= (place((i{:}j{:}k{+}1)) - place((i{:}j{:}k)))/(step((i{:}j{:}k{+}1)) - step((i{:}j{:}k))) \\ &= (0,0) \end{aligned}$$

Variables $c_{i,j}$ stay stationary during the computation. By Theorem 2, *flow* is well-defined.

With functions *step*, *place*, and *flow*, we derive the initial data layout as follows:

$$\begin{aligned} pattern(a_{i,k}) &= place((i{:}j{:}k)) - step((i{:}j{:}k)) * flow(a_{i,k}) \\ &= (i,-i-k) \end{aligned}$$

$$\begin{aligned} pattern(b_{k,j}) &= place((i{:}j{:}k)) - step((i{:}j{:}k)) * flow(b_{k,j}) \\ &= (-j-k,j) \end{aligned}$$

$$\begin{aligned} pattern(c_{i,j}) &= place((i{:}j{:}k)) - step((i{:}j{:}k)) * flow(c_{i,j}) \\ &= (i,j) \end{aligned}$$

By Theorem 3, *pattern* is well-defined.

The network of processors and the initial data layout, as produced by our graphics system, is depicted in Figure 1. Each dot represents an inner product step processor. Arrows represent the propagation of data. A variable name labelling an arrow indicates the location of that variable. If the arrow points to a processor, this variable is input to that processor at the current step of the systolic execution.

The processor layout of this design mirrors matrix C. The number of processors depends on the size of the input. For matrices of large size, this design may require a large number of processors. We can improve this situation by proposing a different place function.

6.1.2. The Second Design

Let us assume that we will keep the band widths of the input matrices constant. That is, when increasing the size of the input, we never widen the matrices' bands. Under this assumption, we can derive for the same matrix multiplication program another design whose number of processors is constant. We must simply find a place function whose coordinates depend only on the band widths of the input matrices but not on their size. The band widths of the input matrices are determined by the differences of i and k and of j and k (see the enabling condition of our neutrality declaration). We choose our coordinates from these differences:

$$place((i{:}j{:}k)) =_{\text{def}} (i-k, j-k)$$

Again, other choices are possible. By Theorem 1, this place function also satisfies (P1):

$$\begin{vmatrix} 1 & 1 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \end{vmatrix} = 3$$

With the new proposed function, we derive the following *flow* and *pattern*:

$$\begin{aligned} flow(a_{i,k}) &= (0,1) \\ flow(b_{k,j}) &= (1,0) \\ flow(c_{i,j}) &= (-1,-1) \end{aligned}$$

$$\begin{aligned} pattern(a_{i,k}) &= (i-k, -i-2k) \\ pattern(b_{k,j}) &= (-j-2k, j-k) \\ pattern(c_{i,j}) &= (2i+j, i+2j) \end{aligned}$$

Flow and *pattern* are, again, well-defined.

The network of processors and the initial data layout is depicted in Figure 2. This design is presented in [10]. The number of processors is $(p_A+q_A+1)*(p_B+q_B+1)$. It is independent of the size of the input.

After arriving at an improved processor layout, we now modify the program to improve execution speed. We could have proceeded in the converse order.

6.1.3. The Third Design

Recall that any two inner product steps are commutative. In Sect. 4.1, we decided not to declare this commutativity. A search reveals that a commutation in the definition of refinement *inner-product* yields the shortest trace:

	inner-product$(i,j,0)$:	$(i{:}j{:}0)$
$\{0<k\}$	*inner-product*(i,j,k):	$(i{:}j{:}k)$; *inner-product*$(i,j,k-1)$

The parallel trace obtained for the multiplication of two 4×4 matrices (*matrix-matrix*(4)) expands to:

```
  <>
→ <>
→ <(0:0:1)>
→ <(0:0:0) (0:1:1) (1:0:1) (1:1:2)>
→ <(0:1:0) (0:2:1) (1:0:0) (1:1:1) (1:2:2) (2:0:1) (2:1:2) (2:2:3)>
→ <(1:1:0) (1:2:1) (1:3:2) (2:1:1) (2:2:2) (2:3:3) (3:1:2) (3:2:3)>
→ <(2:2:1) (2:3:2) (3:2:2) (3:3:3)>
→ <(3:3:2)>
→ <>
→ <>
```

If we do not consider band width, i.e., do not exploit neutrality, this trace has the same length as previous trace: 10 or, in general, $3n-2$. But, contrary to the previous trace, a consideration of band width can shorten this trace: the leading and trailing empty parallel commands result from the elimination of neutral basic statements. Not counting the empty parallel commands, this trace has length 6 or, in general, $n+\min(p_A,q_B)+\min(q_A,p_B)$. Hence, for constant band width and large n, we achieve a speed-up by a factor of 3. The effect of the commutation in *inner-product* is that, in the execution, k is counted down, not up. Therefore, the derived step function contains a subtraction rather than an addition of k:

$$step((i{:}j{:}k)) \;=\; i+j-k$$

The step value of the first (non-empty) parallel command is -1 or, in general, $-\min(p_A,q_B)$. We keep the place function of the second design:

$$place((i{:}j{:}k)) \;=_{\text{def}}\; (i-k,j-k)$$

Again, we derive well-defined flow and pattern functions:

$$\begin{aligned}
flow(a_{i,k}) &= (0,1)\\
flow(b_{k,j}) &= (1,0)\\
flow(c_{i,j}) &= (1,1)\\
pattern(a_{i,k}) &= (i-k,-i-\min(p_A,q_B))\\
pattern(b_{k,j}) &= (-j-\min(p_A,q_B),j-k)\\
pattern(c_{i,j}) &= (-j-\min(p_A,q_B),-i-\min(p_A,q_B))
\end{aligned}$$

Note that *pattern* depends on the band width because the value of the first step does.

The network of processors and the initial data layout (at the first inner product step) is depicted in Figure 3. This design is also presented in [24].

6.2. LU-Decomposition

The problem is to decompose an $n \times n$ matrix A into two other $n \times n$ matrices L and U, such that

$$a_{i,j} = \sum_{k=0}^{n-1} (l_{i,k} * u_{k,j}) \qquad \text{for} \quad 0 \leq i \leq n-1 \text{ and } 0 \leq j \leq n-1$$

That is, matrix A is the product of matrices L and U. The result matrices are of the following form: the elements of the upper triangle of L and the lower triangle of U (excluding the diagonals) are 0, and the elements of the diagonal of L are 1.

For the refinement of this problem, we shall use several different basic statements. One basic statement produces the diagonal elements of matrix U:

$$u_{i,i} := a_{i,i}^{-1}$$

We denote this basic statement by invr(i). Another basic statement produces the upper-triangle elements of matrix U:

$$u_{i,j} := a_{i,j}$$

We denote this basic statement by up(i,j). Our program shall only apply it with arguments i and j such that $i<j$. A third basic statement produces the lower-triangle elements of matrix L:

$$l_{i,j} := a_{i,j} * u_{j,j}$$

We denote this basic statement by lo(i,j). Our program shall only apply it with arguments i and j such that $j<i$. Finally, there is the inner product step, ips($l_{i,k}$,$-u_{k,j}$,$a_{i,j}$), which we denote, in short, by (i:j:k) as in the previous section.

With these basic statements, the following program performs LU-decomposition; matrices L and U are assumed to be initially everywhere zero:

```
for $i$ from 0 to $n-1$ do
    for $j$ from 0 to $n-1$ do
        for $k$ from 0 to min($i,j$) do
            if $i=j \wedge i=k$
              then invr($i$)
              else if $i<j \wedge i=k$
                     then up($i,j$)
                     else if $j<i \wedge j=k$
                            then lo($i,j$)
                            else ($i$:$j$:$k$)
```

Translated to our programming language, this program becomes:

lu-decom(n): *decom*($n-1$,$n-1$)

	$decom(0,m)$:	$row(0,m)$
$\{0<i\}$	$decom(i,m)$:	$decom(i-1,m)$; $row(i,m)$
	$row(i,0)$:	$inner\text{-}product(i,0,0)$
$\{0<j\}$	$row(i,j)$:	$row(i,j-1)$; $inner\text{-}product(i,j,\min(i,j))$
	$inner\text{-}product(i,j,0)$:	$select(i,j,0)$
$\{0<k\}$	$inner\text{-}product(i,j,k)$:	$inner\text{-}product(i,j,k-1)$; $select(i,j,k)$
$\{i=j \wedge i=k\}$	$select(i,j,k)$:	$\mathrm{invr}(i)$
$\{i<j \wedge i=k\}$	$select(i,j,k)$:	$\mathrm{up}(i,j)$
$\{j<i \wedge j=k\}$	$select(i,j,k)$:	$\mathrm{lo}(i,j)$
$\{i\neq k \wedge j\neq k\}$	$select(i,j,k)$:	$(i{:}j{:}k)$

We now enumerate the declarations of idempotence, neutrality, commutativity, and independence of the four basic statements. Even though numerous, the sixteen declarations are derived as straightforwardly as the two declarations for the single basic statements of the matrix multiplication program. When we introduced statement ips, we decided not to consider its idempotence or commutativity. We must now consider the idempotence or commutativity of invr, up, and lo. Invr, up, and lo are always idempotent. They are also, just as ips, always commutative with themselves and, just as for ips, we are not interested in this commutativity. All other commutativity of invr, up, and lo is implied by independence:

(1) idem $\mathrm{invr}(i)$

(2) idem $\mathrm{up}(i,j)$

(3) idem $\mathrm{lo}(i,j)$

(4) $p<j-i \implies$ ntr $\mathrm{up}(i,j)$

(5) $q<i-j \implies$ ntr $\mathrm{lo}(i,j)$

(6) $p<j-k \vee q<i-k \implies$ ntr $(i{:}j{:}k)$

(7) $i_0\neq i_1 \implies \mathrm{invr}(i_0)$ ind $\mathrm{invr}(i_1)$

(8) $i_0\neq i_1 \vee i_0\neq j_1 \implies \mathrm{invr}(i_0)$ ind $\mathrm{up}(i_1,j_1)$

(9) $i_0\neq j_1 \implies \mathrm{invr}(i_0)$ ind $\mathrm{lo}(i_1,j_1)$

(10) $(i_0\neq i_1 \vee i_0\neq j_1) \wedge (i_0\neq k_1 \vee i_0\neq j_1) \implies \mathrm{invr}(i_0)$ ind $(i_1{:}j_1{:}k_1)$

(11) $i_0\neq i_1 \vee j_0\neq j_1 \implies \mathrm{up}(i_0,j_0)$ ind $\mathrm{up}(i_1,j_1)$

(12) $(i_0\neq i_1 \vee j_0\neq j_1) \wedge (i_0\neq j_1 \vee j_0\neq j_1) \implies \mathrm{up}(i_0,j_0)$ ind $\mathrm{lo}(i_1,j_1)$

(13) $(i_0\neq i_1 \vee j_0\neq j_1) \wedge (i_0\neq k_1 \vee j_0\neq j_1) \implies \mathrm{up}(i_0,j_0)$ ind $(i_1{:}j_1{:}k_1)$

(14) $j_0\neq j_1 \implies \mathrm{lo}(i_0,j_0)$ ind $\mathrm{lo}(i_1,j_1)$

(15) $(i_0 \neq i_1 \vee j_0 \neq j_1) \wedge (i_0 \neq i_1 \vee j_0 \neq k_1) \wedge (j_0 \neq j_1 \vee j_0 \neq k_1) \quad \Rightarrow \quad \mathrm{lo}(i_0,j_0)$ ind $(i_1{:}j_1{:}k_1)$

(16) $(i_0 \neq i_1 \vee j_0 \neq j_1) \wedge (i_0 \neq i_1 \vee k_0 \neq k_1) \wedge (j_0 \neq j_1 \vee k_0 \neq k_1) \quad \Rightarrow \quad (i_0{:}j_0{:}k_0)$ ind $(i_1{:}j_1{:}k_1)$

where p_A and q_A are the band widths of the upper and the lower triangles of matrix A, respectively. Actually, in the case of LU-decomposition, none of the idempotences will be exploited by *transform*.

Again, we derive a sequential trace by replacing ";" with "→". For example, the sequential trace for the LU-decomposition of a 4×4 matrix (*lu-decom*(4)) expands to:

```
invr(0)→up(0,1)→up(0,2)→up(0,3)→
         lo(1,0)→(1:1:0)→invr(1)→
                          (1:2:0)→up(1,2)→
                                   (1:3:0)→up(1,3)→
                  lo(2,0)→(2:1:0)→lo(2,1)→
                                   (2:2:0)→(2:2:1)→invr(2)→
                                            (2:3:0)→(2:3:1)→up(2,3)→
                           lo(3,0)→(3:1:0)→lo(3,1)→
                                            (3:2:0)→(3:2:1)→lo(3,2)→
                                                     (3:3:0)→(3:3:1)→(3:3:2)→invr(3)
```

The parallel trace is obtained by letting *transform* exploit the previous declarations of neutrality and independence on this sequential trace. The ten commands of the parallel trace with neutral atoms collect the inner process steps of the previous sequential trace by columns:

```
  <invr(0)>
→ <up(0,1) lo(1,0)>
→ <up(0,2) (1:1:0) lo(2,0)>
→ <up(0,3) invr(1) (1:2:0) (2:1:0) lo(3,0)>
→ <up(1,2) (1:3:0) lo(2,1) (2:2:0) (3:1:0)>
→ <up(1,3) (2:2:1) (2:3:0) lo(3,1) (3:2:0)>
→ <invr(2) (2:3:1) (3:2:1) (3:3:0)>
→ <up(2,3) lo(3,2) (3:3:1)>
→ <(3:3:2)>
→ <invr(3)>
```

As band width, let us assume p_A=2 and q_A=2. After removal of the neutral atoms, we obtain the final parallel trace:

```
  <invr(0)>
→ <up(0,1) lo(1,0)>
→ <up(0,2) (1:1:0) lo(2:0)>
→ <invr(1) (1:2:0) (2:1:0)>
→ <up(1,2) lo(2,1) (2:2:0)>
→ <up(1,3) (2:2:1) lo(3,1)>
→ <invr(2) (2:3:1) (3:2:1)>
→ <up(2,3) lo(3,2) (3:3:1)>
→ <(3:3:2)>
→ <invr(3)>
```

We derive the step function as in the first design of matrix multiplication. This time we formulate *step* for all three basic statements:

$$\begin{aligned}
step(\mathrm{invr}(i)) &= \alpha_{0,0}*i+\alpha_{0,1}\\
step(\mathrm{up}(i,j)) &= \alpha_{1,0}*i+\alpha_{1,1}*j+\alpha_{1,2}\\
step(\mathrm{lo}(i,j)) &= \alpha_{2,0}*i+\alpha_{2,1}*j+\alpha_{2,2}\\
step((i{:}j{:}k)) &= \alpha_{3,0}*i+\alpha_{3,1}*j+\alpha_{3,2}*k+\alpha_{3,3}
\end{aligned}$$

Choosing 0 as the first step value and substituting basic statements of the parallel trace, we obtain the following linear equations:

$$\begin{aligned}
step(\mathrm{invr}(0)) &= \alpha_{0,1} = 0\\
step(\mathrm{invr}(1)) &= \alpha_{0,0}+\alpha_{1,1} = 3\\
step(\mathrm{up}(0,1)) &= \alpha_{1,1}+\alpha_{1,2} = 1\\
step(\mathrm{up}(0,2)) &= 2\alpha_{1,1}+\alpha_{1,2} = 2\\
step(\mathrm{up}(1,2)) &= \alpha_{1,0}+2\alpha_{1,1}+\alpha_{1,2} = 4\\
step(\mathrm{lo}(1,0)) &= \alpha_{2,0}+\alpha_{2,2} = 1\\
step(\mathrm{lo}(2,0)) &= 2\alpha_{2,0}+\alpha_{2,2} = 2\\
step(\mathrm{lo}(2,1)) &= 2\alpha_{2,0}+\alpha_{2,1}+\alpha_{2,2} = 4\\
step((1{:}1{:}0)) &= \alpha_{3,0}+\alpha_{3,1}+\alpha_{3,3} = 2\\
step((1{:}2{:}0)) &= \alpha_{3,0}+2\alpha_{3,1}+\alpha_{3,3} = 3\\
step((2{:}1{:}0)) &= 2\alpha_{3,0}+\alpha_{3,1}+\alpha_{3,3} = 3\\
step((2{:}2{:}1)) &= 2\alpha_{3,0}+2\alpha_{3,1}+\alpha_{3,2}+\alpha_{3,3} = 5
\end{aligned}$$

Solving these equations establishes the following step function:

$$\begin{aligned}
step(\mathrm{invr}(i,j)) &= 3i\\
step(\mathrm{up}(i,j)) &= 2i+j\\
step(\mathrm{lo}(i,j)) &= i+2j\\
step((i{:}j{:}k)) &= i+j+k
\end{aligned}$$

The solution is consistent for the equations obtained by applying the step function to the rest of the basic statements. We observe that *step*(invr(i)), *step*(up(i,j)), and *step*(lo(i,j)) follow from *step*((i:j:k)):

$$\begin{aligned}
step(\mathrm{invr}(i,j)) &= step((i{:}j{:}k)) && \text{for } i{=}j \text{ and } i{=}k\\
step(\mathrm{up}(i,j)) &= step((i{:}j{:}k)) && \text{for } i{<}j \text{ and } i{=}k\\
step(\mathrm{lo}(i,j)) &= step((i{:}j{:}k)) && \text{for } j{<}i \text{ and } j{=}k
\end{aligned}$$

That is, we have formulated *step* collectively for refinement *select* - rather than individually for the four statements *select* applies.

If *step* can be formulated this way, we can also define *place* in terms of just one statement.[8] In the choice of a place function, we draw from our experience with the previous example:

$$place(select(i{:}j{:}k)) =_{\mathrm{def}} (i-k,j-k)$$

Formulated for our four basic statements individually, this becomes:

[8] Actually, we could even have proposed the program without a refinement of *select*, i.e., with *select* as the single basic statement. But that would suggest that all processors have the capacity of executing everyone of the four operations ips, invr, up, and lo, which is an unnecessary complication, in this case.

$$\begin{aligned}
place(\mathrm{invr}(i)) &= (0,0)\\
place(\mathrm{up}(i,j)) &= (0,j-i)\\
place(\mathrm{lo}(i,j)) &= (i-j,0)\\
place(i{:}j{:}k)) &= (i-k,j-k)
\end{aligned}$$

We apply Theorem 1 again to test the consistency of *step* and *place*:

$$\begin{vmatrix} 1 & 1 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \end{vmatrix} = 3$$

The technique of reducing the different types of basic statement of a program into a single type not only enhances the significance of our theorems but also reduces the problem of proposing processor layouts substantially. For another example of its use, see [8].

For the derivation of *flow*, we may choose any pair of applications of refinement *select*. We choose a pair of inner product steps. Variable $l_{i,k}$ appears in inner product steps $(i{:}j{:}k)$ and $(i{:}j{+}1{:}k)$, and these two statements are executed in consecutive steps. Therefore, we can derive the flow of $l_{i,k}$:

$$\begin{aligned}
flow(l_{i,k}) &= (place((i{:}j{+}1{:}k))-place((i{:}j{:}k)))/(step((i{:}j{+}1{:}k))-step((i{:}j{:}k)))\\
&= (0,1)
\end{aligned}$$

Variable $u_{k,j}$ appears in both $(i{:}j{:}k)$ and $(i{+}1{:}j{:}k)$. The flow of $u_{k,j}$ is derived as:

$$\begin{aligned}
flow(u_{k,j}) &= (place((i{+}1{:}j{:}k))-place((i{:}j{:}k)))/(step((i{+}1{:}j{:}k))-step((i{:}j{:}k)))\\
&= (1,0)
\end{aligned}$$

Variable $a_{i,j}$ appears in basic statements $(i{:}j{:}k)$ and $(i{:}j{:}k{+}1)$, and these two statements are executed in consecutive steps. Therefore, we can derive the flow of $a_{i,j}$:

$$\begin{aligned}
flow(a_{i,j}) &= (place((i{:}j{:}k{+}1))-place((i{:}j{:}k)))/(step((i{:}j{:}k{+}1))-step((i{:}j{:}k)))\\
&= (-1,-1)
\end{aligned}$$

By Theorem 2, *flow* is well-defined.

With functions *step*, *place*, and *flow*, we derive the initial data layout:

$$\begin{aligned}
pattern(a_{i,j}) &= place((i{:}j{:}k))-step((i{:}j{:}k))*flow(a_{i,j})\\
&= (2i+j,i+2j)
\end{aligned}$$

$$\begin{aligned}
pattern(u_{k,j}) &= place((i{:}j{:}k))-step((i{:}j{:}k))*flow(u_{k,j})\\
&= (-j-2k,j-k)
\end{aligned}$$

$$\begin{aligned}
pattern(l_{i,k}) &= place((i{:}j{:}k))-step((i{:}j{:}k))*flow(l_{i,k})\\
&= (i-k,-i-2k)
\end{aligned}$$

By Theorem 3, *pattern* is well-defined.

The network of processors and the initial data layout is depicted in Figure 4. The invr processor is represented by a square, up processors are represented by right-pointing triangles, and lo processors are represented by up-pointing triangles. As before, inner product step processors are represented by dots.

Variable values $u_{k,j}$ leaving the network to the right, and $l_{i,k}$ leaving the network to the top are the output of the computation.

This systolic design is inefficient in several respects. Firstly, output data leave the network several steps after they have been produced: once produced by invr and up, $u_{k,j}$ is not altered by lo and the inner product steps which receive it; similarly, once produced by lo, $l_{i,k}$ is not altered. One improvement is to "tap" invr, up, and lo processors with extra channels that extract their outputs from the network immediately. Secondly, negations of upper-triangle elements of U are performed individually by each inner product step. To save the unnecessary operations, we could add temporary variables to the program. According changes result in the classical systolic design for LU-decomposition [10].

7. Evaluation

Let us review how we develop systolic executions and designs. We provide a program (in form of a refinement) and a processor layout (in form of a place function). Given to us are properties of the programming language (in form of semantic relations) and restrictions on the architecture (implicit in the requirements on *step*, *place*, *flow*, and *pattern*). From this information, we synthesize, via a sequential execution, a parallel execution of the program and, via a step function, the data layout and movement (in form of a flow function and a pattern function). We could also exchange what we propose and derive. For example, if we proposed the data movement, we could synthesize the data layout and the processor layout.

Our work is distinguished by the combination of three factors. Embedding systolic design into a general view of programming enables us to separate distinct concerns properly. The explicit formulation of a parallel execution provides a precise link between the two components proposed by the human in a systolic design: the program and the processor layout. Our insistence on formal rigor at every stage expedites the automation of a large part of the development. Theorems aid the human in his part of the development. The systolic design at which we arrive can be informally (graphically) conveyed to the human, but it also has a precise mathematical description.

These benefits are demonstrated by our graphics implementation. As a consequence of the isolation of different development stages (program, execution, architecture) in our method, we can quickly and easily change different parameters, one at a time, and obtain a clear display of the effect on the systolic design.

The pairing of a program with a processor layout makes the evaluation of a design particularly convenient: the program determines the execution speed (as the length of the parallel trace) and the processor layout determines the size of the design (as the number of processors). The density of the data layout is determined only by the pair but not by either component alone. For example, our first and second designs of matrix multiplication are based on the same program but the densities of their data layouts differ. Similarly, our second and third designs have the same processor layout, but the densities of their data layouts differ.

At present, we use *transform* as a heuristic. Our initial definition of it removed neutral elements first, not last. In some cases, this version of *transform* leads to faster executions. We still abandoned it, because it also leads to more complicated step functions, and simplicity is important to us. *Transform* is just another variable in our method. So far, our specific transformation strategy has served us remarkably well.

We are not very satisfied with the way in which we identified the commutation in the definition of *inner-product* that led to our third design for matrix multiplication. We also attempted commutations in the other refinements, *product* and *row*, but they lead to executions that are never shorter and sometimes longer. All we can provide at this time is an implemented system that lets us conduct these searches conveniently. The fact that all statements of the matrix multiplication program are commutative is discouraging. It provides us with no information of what execution to pick.

To reach the first step of our parallel systolic execution, several steps of "soaking up" data may have to be taken. Similarly, after the last step of our execution, data remaining in the network may have to be "drained". After arriving at a particular design, we can compute the lengths of the soaking and draining phases from *step* and *place*. Soaking and draining influences the performance of the design.

Our method is particularly suitable for a search of different systolic designs for some fixed problem. An impressive example is our treatment of the Algebraic Path Problem [8]. The Algebraic Path Problem subsumes many matrix computation problems, among them matrix inversion, transitive closure, and shortest paths. Its solutions are complicated systolic designs with seven different types of operations and different data items being reflected in different directions up to four times on their path through the processor array [22]. For variables whose flow is not constant over the entire execution, the well-definedness of *flow* and *pattern* is violated. However, we can cope with such cases in an incremental fashion. We can extend the parallel execution with statements that copy variables (whose direction of flow changes) to new variables (at the points of change). The flow of each of the resulting variables is then constant. We have obtained an algorithm by which the parallel execution can be successively enhanced with such reflection operations [8].

Programs lend themselves to a systolic implementation if they combine a few simple operations in a highly repetitive way. We expect our method of systolic design to work best for problems in which the program does not reflect aspects of the systolic architecture. That is not the case, for instance, if particular synchronizing or signal processing operations are required in the program as in Snepscheut's systolic design for transitive closure [23]. However, at least in the treatment of the Algebraic Path Problem, we were able to add operations imposed by the architecture at a later stage. Our method works the better, the fewer types of basic operations need to be considered. Many different types of processors can cause an explosion in the number of semantic declarations.

Many researchers have investigated methods of systolic design in recent years (see the next section). All these methods require two kinds of input: one component that can be thought of as a program, and one component that gives some clue about the structure of the systolic array. In our approach both these inputs need not be cleverly chosen. Of the program, we require only that it solve the numerical problem at hand. For the place function, we can start with a simple proposition that looks promising. After evaluating the result of our inputs, we can make incremental variations. These variations may be random, or they may be carefully selected. In our matrix multiplication example, we adjusted each of the two inputs once.

8. Related Research

Chen [5, 6] chooses the inverse of our derivation. She supplies a "network", which is the analogue of our flow function, and an "abstract process structure" (a set of recurrence equations), which is the analogue of our refinement. Her informal derivation results in a "concrete structure", which is the analogue of our step and place functions. Chen does not spell out systolic executions, as we do with traces, and is, in general, less formal.

Like us, Moldovan and Fortes [20] require the input of a program, but their program must be augmented with "artificial" variables [19]. This augmentation is meant to specify parallelism and corresponds roughly to our semantic relations - except that semantic relations are properties of the programming language, not properties of individual programs. (The detection of parallelism receives more attention in another of their papers [7].) Systolic arrays are described by a space transformation which corresponds to our function *place* and a time transformation which corresponds to our function *step*. Moldovan and Fortes require the input of both transformations, while we only require the input of *place* (or even only part of *place*). Similarly to Chen, Moldovan and Fortes present an algorithm by which the space transformation can be derived from a set of proposed flow vectors. Mirankler and Winkler [18] employ the same space-time transformation as Moldovan but use a graph representation.

Chandy and Misra [4] propose an "invariant", which corresponds to our step function, and, with some additional assumptions, derive a systolic program from it. A program in their language, Unity [3], is a repeating multiple assignment statement. Chandy and Misra envision Unity as a tool in which programming solutions for many different architectures can be expressed with equal convenience. They equate the

Unity programs that they derive for matrix multiplication and LU-decomposition with systolic executions and, indeed, with systolic architectures. An essential aspect of our synthesis method is that we distinguish the three concepts of a program, a trace, and an architecture.

Lam and Mostow [11] employ an implemented method of transformation similar to ours but, again, less precise. They do not treat the layout separately from the program but require program annotations that give a clue about the processor layout ("in place" or "in parallel").

Cappello and Steiglitz [2] describe a method of systolic design by geometric transformation. They derive a first data flow scheme from a sequential program execution. The data flow scheme is expressed geometrically in space-time and is, usually, not well-suited for implementation. It is then improved by geometric transformations proposed by the human. As many other approaches in VLSI theory, this one aims at chip layout, not at programming. Our centerpiece, the parallel execution, is missing.

Leiserson's work [12,13] is in the same spirit as ours. He proposes a number of widely applicable transformations that address aspects of communication and timing in processor networks. The transformations alter the characteristics of processor connections after the layout of processors has been decided. Our method helps the human in the choice of an appropriate processor layout. Leiserson's focus is the conversion of semisystolic into systolic architectures. Semisystolic architectures allow for *broadcasting* and *census*, i.e., the parallel communication of data to and from all processors; systolic architectures do not. Broadcasting and census are abstractions that can lead to simpler designs. Our solutions of matrix computation problems can be easily formulated without broadcasting or census.

Systolic design spans several levels of abstraction, from a specification to a chip layout. The two ends of this spectrum are, at present, best understood. The front end is the refinement of a specification into an abstract program. Solutions to this end are offered by work in programming methodology. The back end is the refinement of an abstract systolic architecture into an optimized concrete one. Solutions to this end are offered by work in VLSI design. Our work provides a connection of both ends: it links an abstract program with an abstract systolic architecture.

References

1. Boyer, R. S., and Moore, J S. *A Computational Logic*. ACM Monograph Series, Academic Press, 1979.

2. Cappello, P. R., and Steiglitz, K. Unifying VLSI Array Design with Linear Transformations of Space-time. In *Advances in Computing Research, Vol. 2: VLSI Theory*, F. P. Preparata, Ed., JAI Press Inc., 1984, pp. 23-65.

3. Chandy, M. Concurrent Programming for the Masses. Proc. 4th Ann. ACM Symp. on Principles of Distributed Computing, 1985, pp. 1-12.

4. Chandy, K. M., and Misra, J. "Systolic Algorithms as Programs". *Distributed Computing 1,* 3 (1986), 177-183.

5. Chen, M. C. Synthesizing Systolic Designs. YALEU/DCS/RR-374, Department of Computer Science, Yale University, Mar., 1985.

6. Chen, M. C. A Parallel Language and Its Compilation to Multiprocessor Machines or VLSI. Proc. 13th Ann. ACM Symp. on Principles of Programming Languages, 1986, pp. 131-139.

7. Fortes, J. A. B., and Moldovan, D.I. "Parallelism Detection and Transformation Techniques for VLSI Algorithms". *Journal of Parallel and Distributed Computing 2,* 3 (Aug. 1985), 277-301.

8. Huang, C.-H., and Lengauer, C. An Incremental, Mechanical Development of Systolic Solutions to the Algebraic Path Problem. TR-86-25, Department of Computer Sciences, The University of Texas at Austin, Dec., 1986.

9. Knuth, D. E. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, 1973. Sect. 5.3.4.

10. Kung, H. T., and Leiserson, C. E. Algorithms for VLSI Processor Arrays. In *Introduction to VLSI Systems*, C. Mead and L. Conway, Eds., Addison-Wesley, 1980. Sect. 8.3.

11. Lam, M. S., and Mostow, J. "A Transformational Model of VLSI Systolic Design". *Computer 18,* 2 (Feb. 1985), 42-52.

12. Leiserson, C. E. Systolic and Semisystolic Design (Extended Abstract). Proc. IEEE Int. Conf. on Computer Design/VLSI in Computers (ICCD '83), 1983, pp. 627-632.

13. Leiserson, C. E., and Saxe, J. B. "Optimizing Synchronous Systems". *Journal of VLSI and Computer Systems 1,* 1 (1983), 41-67.

14. Lengauer, C., and Hehner, E. C. R. "A Methodology for Programming with Concurrency: An Informal Presentation". *Science of Computer Programming 2,* 1 (Oct. 1982), 1-18.

15. Lengauer, C. "A Methodology for Programming with Concurrency: The Formalism". *Science of Computer Programming 2,* 1 (Oct. 1982), 19-52.

16. Lengauer, C., and Huang, C.-H. A Mechanically Certified Theorem about Optimal Concurrency of Sorting Networks. Proc. 13th Ann. ACM Symp. on Principles of Programming Languages, 1986, pp. 307-317.

17. Li, G.-H., and Wah, B. W. "The Design of Optimal Systolic Arrays". *IEEE Trans. on Computers C-34,* 1 (Jan. 1985), 66-77.

18. Miranker, W. L., and Winkler, A. "Spacetime Representations of Computational Structures". *Computing 32,* 2 (1984), 93-114.

19. Moldovan, D. I. "On th Design of Algorithms for VLSI Systolic Arrays". *Proc. IEEE 71,* 1 (Jan. 1983), 113-120.

20. Moldovan, D. I., and Fortes, J. A. B. "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays". *IEEE Trans. on Computers C-35,* 1 (Jan. 1986), 1-12.

21. Quinton, P. Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations. Proc. 11th Ann. Int. Symp. on Computer Architecture, 1984, pp. 208-214.

22. Rote, G. "A Systolic Array Algorithm for the Algebraic Path Problem (Shortest Paths; Matrix Inversion)". *Computing 34*, 3 (1985), 191-219.

23. van de Snepscheut, J. L. A. A Derivation of a Distributed Implementation of Warshall's Algorithm (JAN-113a). CS 8505, Dept. of Mathematics and Computing Science, University of Groningen, 1985.

24. Weiser, U., and Davis, A. A Wavefront Notation Tool for VLSI Array Design. In *VLSI Systems and Computations*, H. T. Kung, B. Sproull, and G. Steele, Eds., Computer Science Press, 1981, pp. 226-234.

Appendix: Proofs

We restate and prove Theorems 1, 2, and 3 of Section 3.

Theorem 1: Let *step* be a linear step function for parallel trace t that satisfies (S1) and (S2). Let *place* be a linear place function for t. *Place* satisfies (P1) if the following equations have the zero vector as the unique solution:

$$\begin{aligned}
&\alpha_{0,0}u_0+\alpha_{0,1}u_1+\ldots+\alpha_{0,r-1}u_{r-1}=0\\
&\alpha_{1,0}u_0+\alpha_{1,1}u_1+\ldots+\alpha_{1,r-1}u_{r-1}=0\\
&\ldots\\
&\alpha_{d,0}u_0+\alpha_{d,1}u_1+\ldots+\alpha_{d,r-1}u_{r-1}=0
\end{aligned}$$

where r is the number of arguments of basic statement s.

Proof:

Place satisfies (P1)

$=$ {conditions (S1), (S2) and (P1)}

for all basic statements $s(x_0,x_1,\ldots,x_{r-1})$ and $s(y_0,y_1,\ldots,y_{r-1})$ in t,

$$\begin{aligned}
&s(x_0,x_1,\ldots,x_{r-1})\neq s(y_0,y_1,\ldots,y_{r-1}) \wedge \mathit{step}(s(x_0,x_1,\ldots,x_{r-1}))=\mathit{step}(s(y_0,y_1,\ldots,y_{r-1}))\\
\Rightarrow\quad &\mathit{place}(s(x_0,x_1,\ldots,x_{r-1}))\neq \mathit{place}(s(y_0,y_1,\ldots,y_{r-1}))
\end{aligned}$$

$=$ {*step* and *place* are linear, and equations (E1) and (E2)}

for all basic statements $s(x_0,x_1,\ldots,x_{r-1})$ and $s(y_0,y_1,\ldots,y_{r-1})$ in t,

$$\begin{aligned}
&s(x_0,x_1,\ldots,x_{r-1})\neq s(y_0,y_1,\ldots,y_{r-1})\\
&\wedge\ \alpha_{0,0}x_0+\alpha_{0,1}x_1+\ldots+\alpha_{0,r-1}x_{r-1}+\alpha_{0,r}=\alpha_{0,0}y_0+\alpha_{0,1}y_1+\ldots+\alpha_{0,r-1}y_{r-1}+\alpha_{0,r}\\
\Rightarrow\quad &(\alpha_{1,0}x_0+\alpha_{1,1}x_1+\ldots+\alpha_{1,r-1}x_{r-1}+\alpha_{1,r},\ \ldots,\ \alpha_{d,0}x_0+\alpha_{d,1}x_1+\ldots+\alpha_{d,r-1}x_{r-1}+\alpha_{d,r})\\
&\neq(\alpha_{1,0}y_0+\alpha_{1,1}y_1+\ldots+\alpha_{1,r-1}y_{r-1}+\alpha_{1,r},\ \ldots,\ \alpha_{d,0}y_0+\alpha_{d,1}y_1+\ldots+\alpha_{d,r-1}y_{r-1}+\alpha_{d,r})
\end{aligned}$$

$=$ {algebraic simplification}

for all basic statements $s(x_0,x_1,\ldots,x_{r-1})$ and $s(y_0,y_1,\ldots,y_{r-1})$ in t,

$$\begin{aligned}
&s(x_0,x_1,\ldots,x_{r-1})\neq s(y_0,y_1,\ldots,y_{r-1})\\
&\wedge\ \alpha_{0,0}x_0+\alpha_{0,1}x_1+\ldots+\alpha_{0,r-1}x_{r-1}+\alpha_{0,r}=\alpha_{0,0}y_0+\alpha_{0,1}y_1+\ldots+\alpha_{0,r-1}y_{r-1}+\alpha_{0,r}\\
\Rightarrow\quad &\alpha_{1,0}x_0+\alpha_{1,1}x_1+\ldots+\alpha_{1,r-1}x_{r-1}+\alpha_{1,r}\neq\alpha_{1,0}y_0+\alpha_{1,1}y_1+\ldots+\alpha_{1,r-1}y_{r-1}+\alpha_{1,r}\\
&\vee\ \ldots\\
&\vee\ \alpha_{d,0}x_0+\alpha_{d,1}x_1+\ldots+\alpha_{d,r-1}x_{r-1}+\alpha_{d,r}\neq\alpha_{d,0}y_0+\alpha_{d,1}y_1+\ldots+\alpha_{d,r-1}y_{r-1}+\alpha_{d,r}
\end{aligned}$$

$=$ {algebraic simplification}

for all basic statements $s(x_0,x_1,\ldots,x_{r-1})$ and $s(y_0,y_1,\ldots,y_{r-1})$ in t,

$$\begin{aligned}
&s(x_0,x_1,\ldots,x_{r-1})\neq s(y_0,y_1,\ldots,y_{r-1})\\
&\wedge\ \alpha_{0,0}(x_0-y_0)+\alpha_{0,1}(x_1-y_1)+\ldots+\alpha_{0,r-1}(x_{r-1}-y_{r-1})=0\\
\Rightarrow\quad &\alpha_{1,0}(x_0-y_0)+\alpha_{1,1}(x_1-y_1)+\ldots+\alpha_{1,r-1}(x_{r-1}-y_{r-1})\neq 0\\
&\vee\ \ldots\\
&\vee\ \alpha_{d,0}(x_0-y_0)+\alpha_{d,1}(x_1-y_1)+\ldots+\alpha_{d,r-1}(x_{r-1}-y_{r-1})\neq 0
\end{aligned}$$

$=$ {predicate calculus}

for all basic statements $s(x_0,x_1,\ldots,x_{r-1})$ and $s(y_0,y_1,\ldots,y_{r-1})$ in t,

$$\begin{aligned}&\alpha_{0,0}(x_0-y_0)+\alpha_{0,1}(x_1-y_1)+\ldots+\alpha_{0,r-1}(x_{r-1}-y_{r-1})=0\\ \wedge\ &\alpha_{1,0}(x_0-y_0)+\alpha_{1,1}(x_1-y_1)+\ldots+\alpha_{1,r-1}(x_{r-1}-y_{r-1})=0\\ \wedge\ &\ldots\\ \wedge\ &\alpha_{d,0}(x_0-y_0)+\alpha_{d,1}(x_1-y_1)+\ldots+\alpha_{d,r-1}(x_{r-1}-y_{r-1})=0\end{aligned}$$

$$\Rightarrow\quad s(x_0,x_1,\ldots,x_{r-1})=s(y_0,y_1,\ldots,y_{r-1})$$

$\Leftarrow$ {algebraic simplification}

$$\begin{aligned}&\alpha_{0,0}u_0+\alpha_{0,1}u_1+\ldots+\alpha_{0,r-1}u_{r-1}=0\\ &\alpha_{1,0}u_0+\alpha_{1,1}u_1+\ldots+\alpha_{1,r-1}u_{r-1}=0\\ &\ldots\\ &\alpha_{d,0}u_0+\alpha_{d,1}u_1+\ldots+\alpha_{d,r-1}u_{r-1}=0\end{aligned}$$

have the zero vector as the unique solution.

(End of Proof)

Theorem 2: Let *step* be a linear step function for parallel trace t that satisfies (S1) and (S2). Let *place* be a linear place function for t that satisfies (P1). If the subscripts of variable v are determined by all but one of the r arguments of the basic statement, then *flow* is well-defined for variable v.

Proof: Let $s_x=s(x_0,\ldots,x_i,\ldots,x_{r-1})$, $s_{x'}=s_x[x_i'/x_i]$, $s_y=s_x[y_i/x_i]$, and $s_{y'}=s_x[y_i'/x_i]$. Let the subscripts of variable v be x_0, ..., x_{i-1}, x_{i+1}, ..., and x_{r-1}, that is, the arguments of basic statement s_x, except the (i+1)-st one, x_i. Then, s_x, $s_{x'}$, s_y, and $s_{y'}$ all access variable $v_{x_0,\ldots,x_{i-1},x_{i+1},\ldots,x_{r-1}}$. Assuming $step(s_x)\neq step(s_{x'})$, and $step(s_y)\neq step(s_{y'})$, we can conclude:

flow is well-defined for variable $v_{x_0,\ldots,x_{i-1},x_{i+1},\ldots,x_{r-1}}$

$=$ {well-definedness}

$(place(s_x)-place(s_{x'}))/(step(s_x)-step(s_{x'}))=(place(s_y)-place(s_{y'}))/(step(s_y)-step(s_{y'}))$

$=$ {*step* and *place* are linear, and s_x, $s_{x'}$, s_y, and $s_{y'}$ have identical arguments in all positions but i}

$(\alpha_{1,i}(x_i-x_i'),\ldots,\alpha_{d,i}(x_i-x_i'))/\alpha_{0,i}(x_i-x_i')=(\alpha_{1,i}(y_i-y_i'),\ldots,\alpha_{d,i}(y_i-y_i'))/\alpha_{0,i}(y_i-y_i')$

$=$ {algebraic simplification}

$(\alpha_{1,i}/\alpha_{0,i},\ldots,\alpha_{d,i}/\alpha_{0,i})=(\alpha_{1,i}/\alpha_{0,i},\ldots,\alpha_{d,i}/\alpha_{0,i})$

$=$ {algebraic simplification}

true

(End of Proof)

Theorem 3: Let *step* be a linear step function for parallel trace t that satisfies (S1) and (S2). Let *place* be a linear place function for t that satisfies (P1). Let *flow*, derived from *step* and *place*, be well-defined. Then *pattern*, derived from *step*, *place*, and *flow*, is well-defined.

Proof: If basic statements *s0* and *s1* are distinct and access variable v of identical subscripts:

pattern is well-defined for variable v

= {well-definedness}

$place(s0)-(step(s0)-fs)*flow(v)=place(s1)-(step(s1)-fs)*flow(v)$

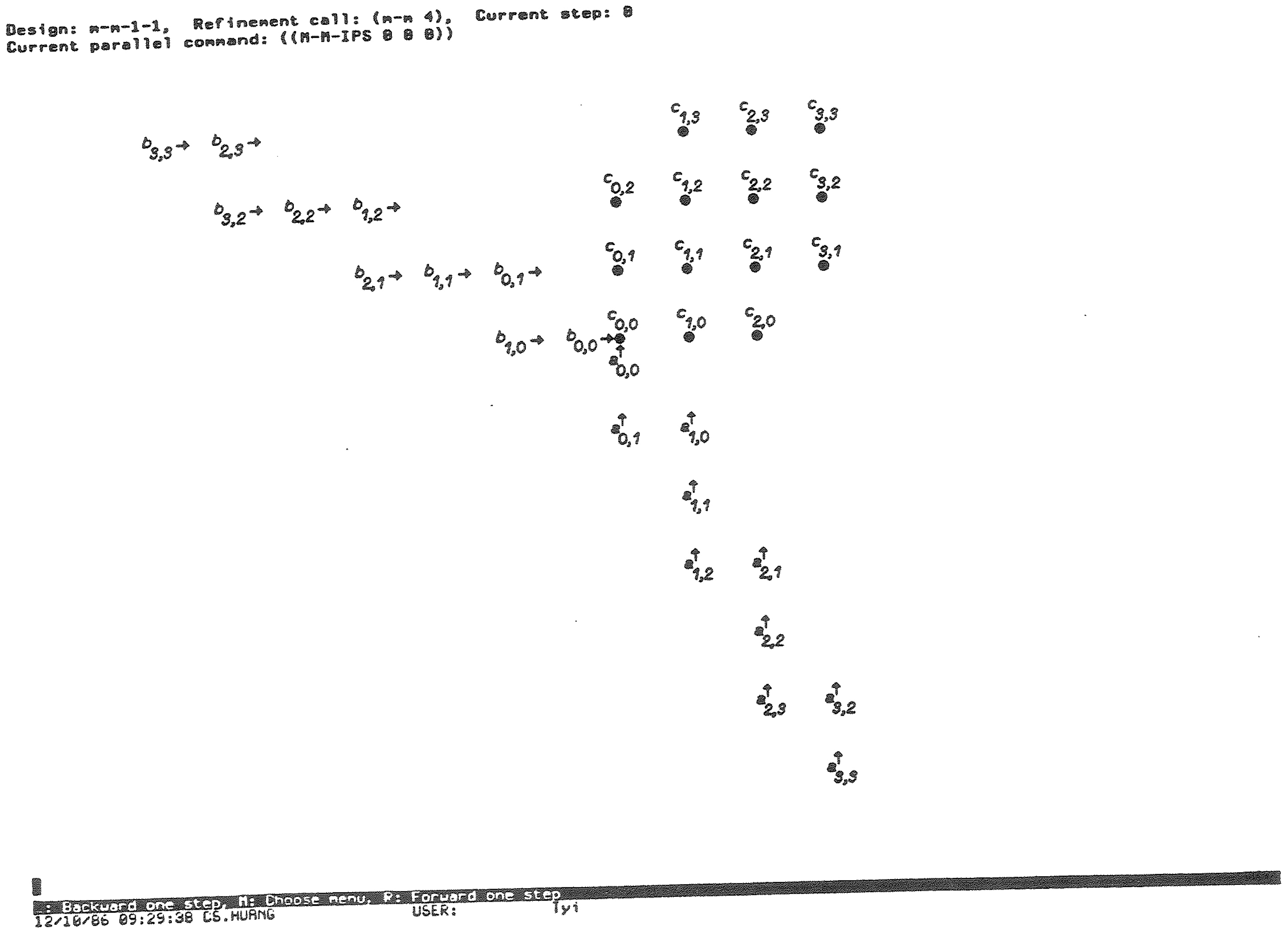
= {algebraic simplification}

$place(s0)-place(s1)=(step(s0)-step(s1))*flow(v)$

= {definition of *flow*}

true

(End of Proof)



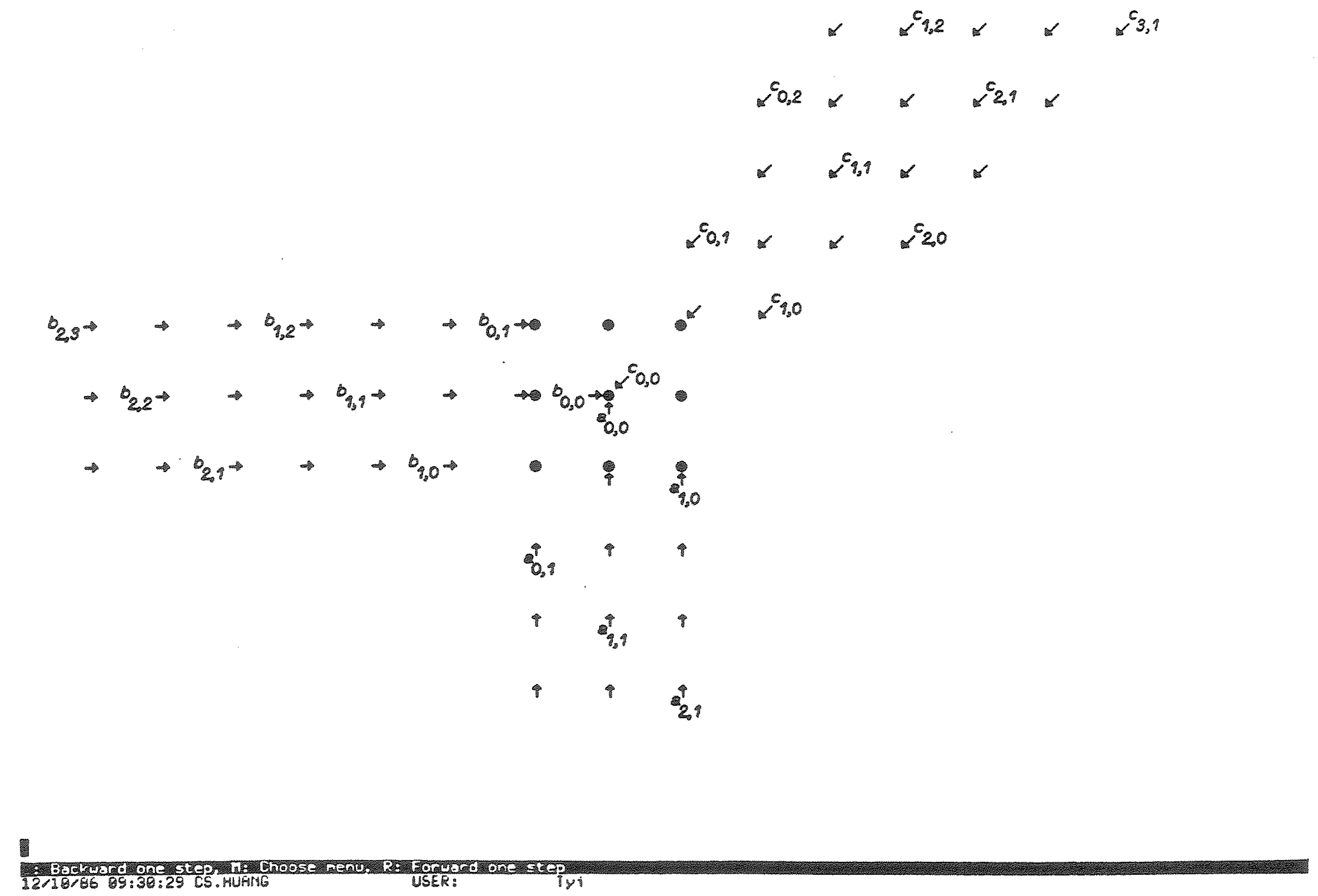
```
Design: m-m-1-1,  Refinement call: (m-m 4),  Current step: 0
Current parallel command: ((M-M-IPS 0 0 0))
```

$b_{3,3}\rightarrow$ $b_{2,3}\rightarrow$
$b_{3,2}\rightarrow$ $b_{2,2}\rightarrow$ $b_{1,2}\rightarrow$
$b_{2,1}\rightarrow$ $b_{1,1}\rightarrow$ $b_{0,1}\rightarrow$
$b_{1,0}\rightarrow$ $b_{0,0}\rightarrow$
$c_{1,3}$ $c_{2,3}$ $c_{3,3}$
$c_{0,2}$ $c_{1,2}$ $c_{2,2}$ $c_{3,2}$
$c_{0,1}$ $c_{1,1}$ $c_{2,1}$ $c_{3,1}$
$c_{0,0}$ $c_{1,0}$ $c_{2,0}$
$a_{0,0}$
$a_{0,1}$ $a_{1,0}$
$a_{1,1}$
$a_{1,2}$ $a_{2,1}$
$a_{2,2}$
$a_{2,3}$ $a_{3,2}$
$a_{3,3}$
: Backward one step, M: Choose menu, P: Forward one step

```
12/10/86 09:29:38 CS.HUANG              USER:         Tyi
```

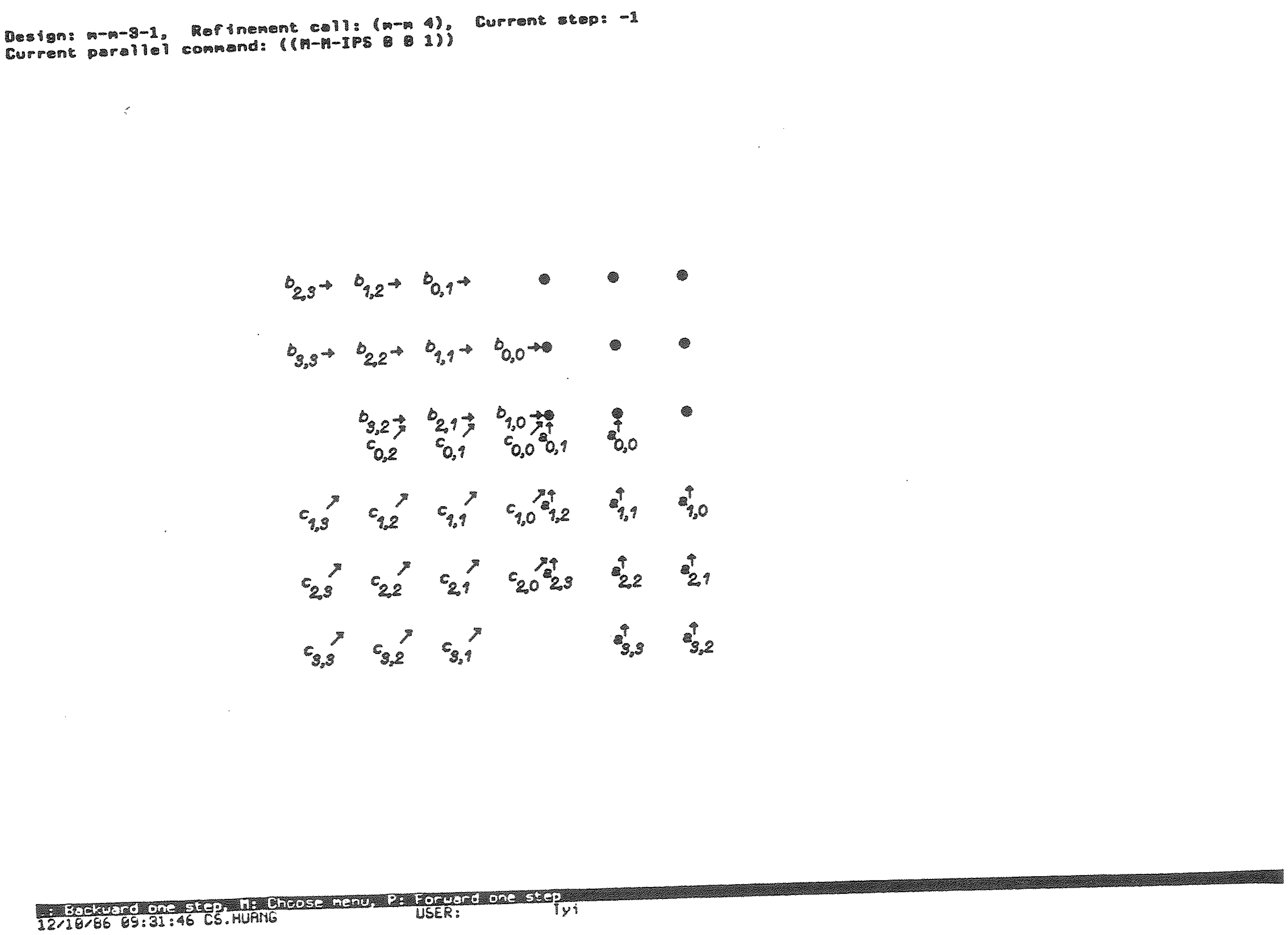
Figure 1. Matrix Multiplication – The First Design

```
Design: m-m-2-1,  Refinement call: (m-m 4),  Current step: 0
Current parallel command: ((M-M-IPS 0 0 0))
```



```
: Backward one step, M: Choose menu, R: Forward one step
12/10/86 09:30:29 CS.HUANG                USER:           Tyi
```

Figure 2. Matrix Multiplication -- The Second Design

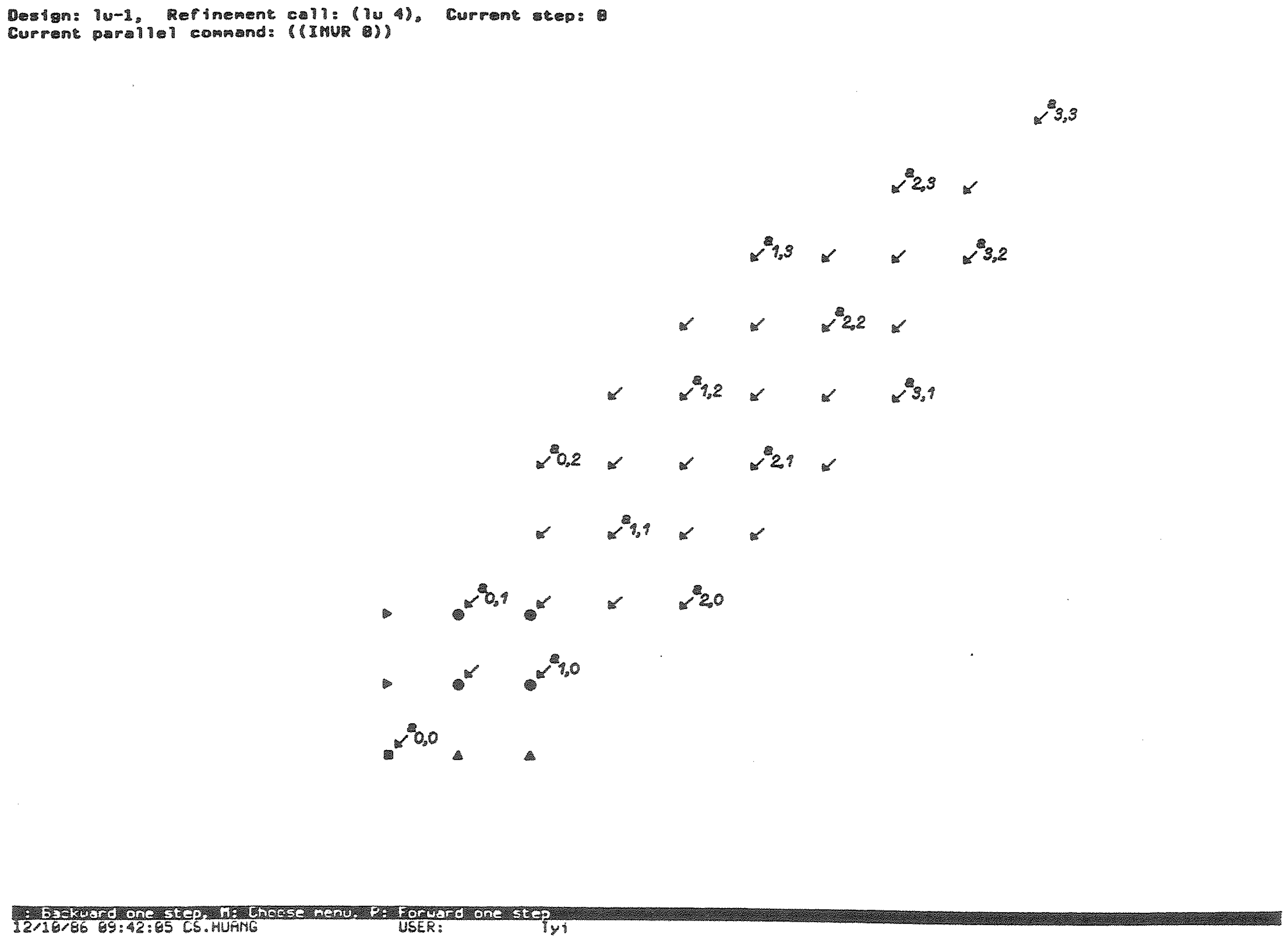


```
Design: m-m-3-1,  Refinement call: (m-m 4),  Current step: -1
Current parallel command: ((M-M-IPS 0 0 1))
```

$b_{2,3}$ $b_{1,2}$ $b_{0,1}$
$b_{3,3}$ $b_{2,2}$ $b_{1,1}$ $b_{0,0}$
$b_{3,2}$ $b_{2,1}$ $b_{1,0}$
$c_{0,2}$ $c_{0,1}$ $c_{0,0}$ $a_{0,1}$ $a_{0,0}$
$c_{1,3}$ $c_{1,2}$ $c_{1,1}$ $c_{1,0}$ $a_{1,2}$ $a_{1,1}$ $a_{1,0}$
$c_{2,3}$ $c_{2,2}$ $c_{2,1}$ $c_{2,0}$ $a_{2,3}$ $a_{2,2}$ $a_{2,1}$
$c_{3,3}$ $c_{3,2}$ $c_{3,1}$ $a_{3,3}$ $a_{3,2}$

```
: Backward one step, M: Choose menu, P: Forward one step
12/10/86 09:31:46 CS.HUANG               USER:           lyi
```

Figure 3. Matrix Multiplication — The Third Design



```
Design: lu-1,  Refinement call: (lu 4),  Current step: 0
Current parallel command: ((INVR 0))
```

```
: Backward one step, M: Choose menu, P: Forward one step
12/10/86 09:42:05 CS.HUANG                USER:          Tyi
```

Figure 4. LU-Decomposition

From: Christian.Lengauer@SAM.CS.CMU.EDU
To: cs.kata@R20.UTEXAS.EDU
Subject: Can you please mail me...

```
...three copies of TR-86-10 to: Dept. of Computer Science
                                Carnegie Mellon University
                                Schenley Park
                                Pittsburgh, PA 15213
```

Thanks.   Chris.

10/07/87

86.10 - 5 cc.

pgs 35