STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLV**, Number 2, 2000

SOME PARALLEL NONDETERMINISTIC ALGORITHMS

VIRGINIA NICULESCU

Abstract. Nondeterminism is useful in two ways. First, it is employed to derive simple and general programs, where the simplicity is achieved by avoiding unnecessary determinism; such programs can be optimized by limiting the nondeterminism. Second, some systems are inherently nondeterministic; programs that represent such systems have to employ some nondeterministic construct. Nondeterministic programs can be mapped more easier on parallel machine, since parallelism brings some nondeterminism by itself.

In this article, there are constructed some nondeterministic programs, for some numerical methods, using the UNITY notation[3]. The correctness of the algorithms is proven, and some possible mappings are discussed.

1. Introduction

Nondeterminism is useful in two ways. First, it is employed to derive simple and general programs, where the simplicity is achieved by avoiding unnecessary determinism; such programs can be optimized by limiting the nondeterminism. Second, some systems are inherently nondeterministic; programs that represent such systems have to employ some nondeterministic construct.

There is a variety of parallel architectures, though parallel programs have to be developed such that they can be mapped in different ways, on different architectures. A solution is to specify little in the early stages of design, and specify enough in the final stages to ensure efficient execution on target architecture. Specifying little about program execution means that the programs may be nondeterministic.

To express the nondeterministic programs, the model used for the developing the programs is UNITY [3]: "Unbounded Nondeterministic Iterative Transformations", which is briefly described in the next section.

2. A Programming Notation

The UNITY program structure is

2000 *Mathematics Subject Classification.* 68N19.

1998 *CR Categories and Descriptors.* G.1.3. [**Mathematics of Computing**] : Numerical Analysis – *Numerical Linear Algebra*; G.4. [**Mathematics of Computing**] : Mathematical Software; D.1.3 [**Software**] : Programming Techniques – *Concurrent Programming.*

$$
\begin{array}{lcll}
program & \rightarrow & Program & program-name \\
 & & declare & declare-section \\
 & & always & always-section \\
 & & initially & initially-section \\
 & & assign & assign-section \\
end & & &
\end{array}
$$

The $declare-section$, names the variables used in the program and their types. The syntax is similar to that used in Pascal. The $always-section$ is used to define certain variables as function of others. This section is not necessary for writing UNITY programs, but it is convenient. The $initially-section$ is used to define initial values of some of the variables; uninitialized variables have arbitrary initial values. The $assign-section$ contains a set of assignment statements.

The program execution starts in a state where the values of variables are as specified in the initially-section. (A state is characterized by the values of all variables.) In each step, any one statement is executed. Statements are selected arbitrarily for execution, though in an infinite execution of the program each statement is executed infinitely often. A state of a program is called a *fixed point* if and only if execution of any statement of the program, in this state, leaves the state unchanged. A predicate, called FP, characterize the fixed points of the program. Once FP holds, continued execution leaves values of all variables unchanged, and therefore it makes no difference whether the execution continues or terminates.

The termination of a program is regarded as a feature of an implementation. A program execution is an infinite sequence of statement executions and an implementation is a finite prefix of the sequence.

2.1. **Mapping Programs to Architectures.** One way to implement a program is to halt it after it reaches a fixed point.

A mapping to a von Neumann machine specifies the schedule for executing assignments and the manner in which a program execution terminates.

In a synchronous shared-memory system, a fixed number of identical processors share a common memory that can be read and written by any processors. The synchronism inherent in a multiple-assignment makes it convenient to map such a statement to this architecture.

A UNITY program can be mapped to asynchronous shared-memory system, by partitioning the statements of the program among the processors. In addition, a schedule of execution for each processor should be specified that guarantees a fair execution for each partition. If the execution for every partition is fair, then any fair interleaving of these executions determines a fair execution of the entire program. Two statements are not executed concurrently if one modifies a variable that the other uses.

Other architectures can be considered for mappings.

2.2. **Assignment Statement.** It is allowed that a number of variables to be assigned simultaneously in a multiple assignment, as in

$$x, y, z := 0, 1, 2.$$

Such an assignment can also be written as a set of assignment-components separated by ||, as in

$$x, y := 0, 1 \| z := 2$$

or

$$x := 0 \| y := 1 \| z := 2.$$

The variables to be assigned and the values to be assigned to them may be described using quantification, rather than enumeration:

$$< \| i : 0 \leq i < N :: A[i] := B[i] > .$$

A notation like the following is used for a conditional assignment:

$$\begin{array}{rl} x := -1 & if\ y < 0\ \sim \\ 0 & if\ y = 1\ \sim \\ 1 & if\ y > 0\ . \end{array}$$

2.3. **Assign-section.** The symbol ‡ acts as a separator between the statements. A quantified-statement-list denotes a set of statements obtained by instantiating the statement-list with the appropriate instances of bounded variables; if there is no instance, quantified-statement-list denotes an empty set of statements. The number of the instances must be finite. The boolean expression in the quantification should no name program variables whose values may change during program execution.

2.4. **Initially-section.** The syntax of this section is the same as that of the assign-section except that symbol := is replaced with =. The equations defining the initial values should not be circular.

2.5. **Always-section.** An always-section is used to define certain program variables as function of other variables. The syntax used in the always-section is the same as in the initially-section.

3. Nondeterministic Gauss Elimination

We consider the Gaussian elimination scheme for solving a set of linear equations,

$$A \cdot X = B,$$

where $A[0..n-1, 0..n-1]$ and $B[0..n-1]$ are given and the solution is to be stored in $X[0..n-1]$. Gaussian elimination is presented typically as a sequence of n pivot steps. The following UNITY program allows nondeterministic choices in the selections of the pivot rows.

3.1. **A Solution.** Let $M(A;B)$ (or M for short) the matrix with n rows and $n+1$ columns, where the first n columns are from A and the last column is from B. In the Gaussian elimination $M(A;B)$ is modified to $M(A^{'};B^{'})$ by certain operations such that

$$A \cdot X = B$$

and

$$A^{'} \cdot X = B^{'}$$

have the same solutions for X. The goal of the algorithm is to apply a sequence of these operations to convert $M(A;B)$ to $M(I_n;X_F)$, where I_n is the identity matrix; then X_F is the desired solution vector. This goal can be realized if the rank of A is n, which we assume to be the case.

The program consists of two kinds of statements:

(1) Pivot with row u, provided that $M[u,u] \neq 0$; this has the effect of setting $M[u,u]$ to 1 and $M[v,u]$ to 0, for all $v, v \neq u$
(2) Exchange two rows u and v, provided that both $M[u,u]$ and $M[v,v]$ are zero and at least one of $M[u,v], M[v,u]$ is nonzero; this has the effect of replacing a zero diagonal with a nonzero element.

Due to the fact that there are some possible exchanges between the rows, the elements of the solution vector will be exchanged also. The permutation of the elements is stored in an array p.

$$
\begin{array}{l}
\mathit{Program\ Gauss} \\
\quad \mathit{declare} \\
\quad\quad M : \mathit{array}[0..n-1, 0..n]\ \mathit{of\ real} \\
\quad\quad p : \mathit{array}[0..n-1]\ \mathit{of\ integer} \\
\quad \mathit{initially} \\
\quad\quad < i : 0 \leq i < n : p[i] = i > \\
\quad \mathit{assign} \\
\quad\quad \{\text{pivot with row } u \text{ if } M[u,u] \neq 0\} \\
\quad\quad < \ddagger u : 0 \leq u < n :: \\
\quad\quad\quad < \| v, j : 0 \leq j < n \wedge 0 \leq v < n \wedge v \neq u :: \\
\quad\quad\quad\quad M[v,j] := M[v,j] - M[v,u] \cdot M[u,j]/M[u,u]\ \mathit{if}\ M[u,u] \neq 0 \\
\quad\quad\quad > \\
\quad\quad\quad \| < \| j : 0 \leq j < n :: \\
\quad\quad\quad\quad M[u,j] := M[u,j]/M[u,u]\ \mathit{if}\ M[u,u] \neq 0 \\
\quad\quad\quad > \\
\quad\quad >
\end{array}
$$

$\ddagger\{$exchange two rows if both have zero diagonal elements and the
exchange results in at least one of these elements being set to nonzero$\}$
$< \ddagger u, v : 0 \leq u < n \wedge 0 \leq v < n \wedge u \neq v ::$
$< \| j : 0 \leq j < n :: M[u,j], M[v,j], p[u], p[v] := M[v,j], M[u,j], p[v], p[u]$
$if\ M[u,u] = 0 \wedge M[v,v] = 0 \wedge (M[u,v] \neq 0 \vee M[v,u] \neq 0)$
$>$
$>$
$end\{Gauss\}$

3.2. **Correctness.** Let M^0 denote the initial Z matrix. Since each statement in the program modifies M such that the solutions to the given linear equations are preserved, we have the following invariant:

$$invariant \quad M^0, M \text{ have the same solution.}$$

In the following, A refers to the $n \times n$ matrix in the left part of M, and B, to the last column of M. First, it is proven that the program *Gauss* reaches a fixed point and that at any fixed point, A is an identity matrix. Then, from the invariant, B is the desired solution vector. In the following, a *unit column* is a column in which the diagonal element is 1 and all other elements are 0. That is, column u is a unit column means that

$$M[v,u] = 0\ if\ u \neq v \ \sim 1\ if\ u = v.$$

To show that a fixed point is reached, it is proven that the pair (p, q), where

$p =$ number of unit columns in A
$q =$ number of nonzero diagonal elements in A,

increases lexicographically with every state change.

We consider each statement in turn. Pivoting with row u, where column u is a unit column, cause no state change. A state change results from a pivot operation with row u only if column u is not a unit column; the effect of the pivot operation is to set u to a unit column, thus increasing p.

Two rows u and v are exchanged only when $M[u,u] = 0 \wedge M[v,v] = 0 \wedge (M[u,v] \neq 0 \vee M[v,u] \neq 0)$. Hence neither of the columns u or v is a unit column. The exchange preserves all the unit columns, also preserving p. In addition, at least one diagonal element, $M[u,u]$ or $M[v,v]$ is set to nonzero. Since both of these elements were previously zero, q increases. Therefore, every state change in program *Gauss* increases (p, q) lexicographically. Since each of p, q is bounded from above by n, *Gauss* reaches a fix point.

Now, it must be proved that A is an identity matrix at any fix point. The proof is as follows. Lemma 1 proves that if any diagonal element $M[u,u]$ is nonzero at a fixed point, u is a unit column. Lemma 2 proves that if some diagonal element is zero at a fix point, all elements in the row are zero. This contradicts the assumption

that the determinant of A is nonzero. (Note that execution of any statement in *Gauss* preserves the determinant.) Therefore every diagonal element is nonzero and, using Lemma 1, A is an identity matrix.

Lemma 1. *At any fixed point of program Gauss,*

$$M[u,u] \neq 0 \Rightarrow u \text{ is a unit column.}$$

Proof: Consider the statement for a pivot corresponding to row u. At any fix point, given that $M[u,u] \neq 0$, for any j and $v, u \neq v$,

$$M[v,j] = M[v,j] - M[v,u] \cdot M[u,j]/M[u,u]$$

and

$$M[u,j] = M[u,j]/M[u,u].$$

In particular, with $j = u$,

$$M[v,u] = M[v,u] - M[v,u] \cdot M[u,u]/M[u,u] = 0$$

and

$$M[u,u] = M[u,u]/M[u,u] = 1.$$

Therefore u is a unit column.

Lemma 2. *At any fixed point of program Gauss,*

$$M[u,u] = 0 \Rightarrow M[u,v] = 0, \forall v \neq u.$$

Proof: Consider two cases: $M[v,v] = 0$ and $M[v,v] \neq 0$.

In the first case, consider the exchange statement for rows u, v. At any fix point, given that $M[u,u] = 0 \wedge M[v,v] = 0$:

$$(M[u,u] = 0 \wedge M[v,v] = 0) \vee (\wedge j :: M[u,j] = M[v,j]).$$

Consider the particular case, $j = v$. Then,

$$(M[u,u] = 0 \wedge M[v,v] = 0) \vee (M[u,v] = M[v,v]).$$

Using the fact that $M[v,v] = 0$ we conclude that $M[u,v] = 0$.

In the second case, if $M[v,v] \neq 0$ from Lemma 1, $M[u,v] = 0$.

3.3. **Mappings.** Program *Gauss* can be implemented in a variety of ways on different architectures. For a sequential machine, it may be more efficient to choose the pivot rows in a particular order. The correctness of this scheme is obvious from the proof because it is obtained from the given program by restricting the nondeterministic choices in statement executions. For an asynchronous shared-memory or distributed architecture, the given program admits several possible implementations; the simplest one is to assign a process to a row. To facilitate the exchange operation, it is possible to allow the row number at a process to be changed. Two rows can be then exchanged simply by exchanging their row numbers. A parallel synchronous architecture with $O(n)$ processors can complete

each exchange operation in a constant time and each pivot operation in $O(n)$ steps; with $O(n^2)$ processors, a pivot operation takes constant time.

4. The Inverse of a Matrix

The method we use for the computation of the inverse matrix use Gauss-Jordan steps. A Gauss-Jordan step with the pivot element $a[u,v] \neq 0$ transforms the matrix A elements, in the following way:

$$a[i,j] = \begin{cases} \frac{1}{a[u,v]} & ,i = u \wedge j = v \\ \frac{-a[i,j]}{a[u,v]} & ,i = u \wedge j \neq v \\ \frac{a[i,j]}{a[u,v]} & ,i \neq u \wedge j = v \\ \frac{a[i,j]\cdot a[u,v]-a[i,v]\cdot a[u,j]}{a[u,v]} & ,i \neq u \wedge j \neq v \end{cases} .$$

If we apply a Gauss-Jordan step n times on matrix $A[0..n-1, 0..n-1]$ we obtain the inverse matrix A^{-1} [2]. We assume that the rank of matrix A is n.

4.1. **A Solution.** The choice of the pivot element it is done in nondeterministic way, provided that it is nonzero. Since, a pivot operation have to be done only one time for a particular row u and a particular column v, after the execution of a pivot operation with the pivot element $a[u,v]$ we set $ind1[u] = 1$ and $ind2[v] = 1$. The $ind1$ and $ind2$ are two arrays which indicate the possible pivot steps. An elimination step with the pivot $a[u,v]$ can be executed only if $ind1[u] = 0 \wedge ind2[v] = 0$.

Because we not choose every time pivot elements from the diagonal, a permutations of the rows of the inverse matrix results. The permutation p depends of the choices of the pivot elements.

Program inverse
declare
  $a : array[0..n-1, 0..n-1]$ *of real*
  $ind1, ind2 : array[0..n-1]$ *of integer*
  $p : array[0..n-1]$ *of integer*
initially
  $< u : 0 \leq u < n :: ind1[u], ind2[u] = 0, 0 >$
  assign
    {pivot operation with the element u, v if $a[u,v] \neq 0$}
    $< \ddagger u, v : 0 \leq u < n \wedge 0 \leq v < n ::$
      $< \| i, j : 0 \leq i < n \wedge 0 \leq j < n ::$

$$
\begin{array}{lll}
a[i,j] & := 1/a[u,v] & if\ i = u \wedge j = v\ \sim \\
 & := -a[u,j]/a[u,v] & if\ i = u \wedge j \neq v\ \sim \\
 & := a[i,v]/a[u,v] & if\ i \neq u \wedge j = v\ \sim \\
 & := (a[i,j] \cdot a[u,v] - a[u,j] \cdot a[i,v])/a[u,v] & if\ i \neq u \wedge j \neq v
\end{array}
$$

$>$

$\|ind1[u], ind2[v], p[u] := 1, 1, v$

$if \quad a[u,v] \neq 0 \wedge ind1[u] = 0 \wedge ind2[v] = 0$

$>$

$end\{inverse\}$

4.2. **Correctness.** If we denote by p the following sum $p = (\sum u : 0 \leq u < n : ind1[u])$, and by q the sum $q = (\sum u : 0 \leq u < n : ind2[u])$, it can be easy proved that the for the pair (p, q) the equality $p = q$ holds at any moment of the execution. So, we can write:

$$invariant \quad p = q.$$

The number $p(p = q)$ increase after the execution of any statement. The values for p and q are bounded from above by n, hence the program $inverse$ reaches at a fix point, where $p = q = n$.

The equality $p = q = n$ which holds at any fix point shows that there are executed exact n Gauss-Jordan steps with pivot elements from different rows and columns. Therefore the matrix A at any fix point is the inverse matrix of the initial matrix, possible with the rows permuted.

To transform the result to the true inverse matrix the following program can be used.

$Program\ transform$
$declare$
$a : array[0..n-1, 0..n-1]\ of\ real$
$p : array[0..n-1]\ of\ iuteger$
$assign$
$< \ddagger u, v : 0 \leq u < n \wedge 0 \leq v < n ::$
{rows exchange }
$< \|j : 0 \leq j < n :: a[u,j], a[v,j] := a[v,j], a[u,j] >$
$\| p[u], p[v] := p[v], p[u]$
$if\ p[u] = v \vee p[v] = u$
$>$
$end\{transform\}$

4.3. **Mappings.** On a sequential architecture the program $inverse$ can be mapped by choosing the first pivot element founded; the search of the element is made depending in $ind1$ and $ind2$.

The program can be implemented on an asynchronous shared-memory system, by assigning a processor to a row, or by assigning a processor to each matrix element (and so the operations associated with it), provided that there are enough processors.

On a parallel synchronous architecture with n^2 processors the execution of the program takes $O(n)$ time.

4.4. **Other Applications.** The program *inverse* can be used to find the rank of a matrix. The rank it will be equal to $p = q$, which represents the number of the Gauss-Jordan steps, which were executed.

With slight modifications, this program can be used to resolve a system of linear equations. The matrix A is replaced with the matrix M defined for the Gauss program $M = [A|B]$ and finally the result (the solution vector)is the last column of the matrix at the fix point. A permutation of the elements it is done in this case also.

The application of n Gauss-Jordan steps represents also the second stage of the algorithm SIMPLEX.

5. Conclusions

There are presented some nondeterministic algorithms from numerical analysis. Their correctness was proved, and different mappings are discussed.

Nondeterministic programs can be mapped more easier on parallel machine, because the parallelism brings some nondeterminism by itself.

Interesting algorithms can be developed using the concept of nondeterminism. Nondeterministic programs can be implemented on different architectures, in efficient ways.

References

[1] G. E. Blelloch, B. M. Maggs , *Parallel Algorithms*, ACM Computing Surveys, Vol. 28, No. 1, March 1996, pg. 51-54.
[2] W.W. Breckner, Operational Research, "Babeş-Bolyai" University, Cluj-Napoca, 1981 (in Romanian).
[3] K.M. Chandy, J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.
[4] Gh. Coman, *Numerical Analysis*, Libris, Cluj-Napoca, 1995 (in Romanian).
[5] I. Foster, Designing and Building Parallel Programs, 1995.
[6] Carrol Morgan, *Programming from Specifications*, Prentice Hall, 1990.

Department of Computer Science, "Babeş-Bolyai" University, RO-3400 Cluj-Napoca, 1 Kogălniceanu St., RO-3400 Cluj-Napoca, Romania
E-mail address: gina@cs.ubbcluj.ro