Science of Computer Programming 13 (1989/90) 133-180
North-Holland

STEPWISE REFINEMENT OF PARALLEL ALGORITHMS

R.J.R. BACK and K. SERE
Åbo Akademi University, Department of Computer Science, SF-20520 Turku, Finland

Communicated by C.B. Jones
Received November 1989

Abstract. The refinement calculus and the action system formalism are combined to provide a uniform method for constructing parallel and distributed algorithms by stepwise refinement. It is shown that the sequential refinement calculus can be used as such for most of the derivation steps. Parallelism is introduced during the derivation by refinement of atomicity. The approach is applied to the derivation of a parallel version of the Gaussian elimination method for solving simultaneous linear equation systems.

1. Introduction

Stepwise refinement is one of the main methods for systematic construction of sequential programs: a high-level specification of a program is transformed by a sequence of correctness preserving transformations into an executable and efficient program that satisfies the original specification. The *refinement calculus* is a formalization of the stepwise refinement approach. It was first described by Back [1, 2] and has been further elaborated in [3, 4, 23, 25].

The *action system* formalism for parallel and distributed computations was introduced by Back and Kurki-Suonio [7, 8]. The behaviour of parallel and distributed programs using this approach is described in terms of actions, which processes in the system carry out in co-operation with each other. Several actions can be executed in parallel, as long as the actions do not have any variables in common. The actions are atomic: if an action is chosen for execution, it is executed to completion without any interference from the other actions in the system.

Atomicity guarantees that a parallel execution of an action system gives the same results as a sequential and nondeterministic execution. This allows us to use the sequential refinement calculus to construct parallel action systems by stepwise refinements. We can start our derivation from a more or less sequential algorithm and successively increase the degree of parallelism in it, while preserving the correctness of the algorithm. We show in this paper how the refinement calculus and the action system formalism are combined in a powerful and uniform method for deriving parallel algorithms by stepwise refinement. Parallelism is introduced, e.g., by refining the atomicity of actions. A method to carry out such a refinement is developed in [5].

The refinement calculus is based on the assumption that the notion of correctness we want to preserve is total correctness. The refinement relation is not bound to the choice of this specific notion of correctness, but much of the methods developed and the theory are specific to this choice. For sequential programs this correctness notion does not require any justification. Many distributed programs are, however, reactive: in addition to the final result also the way in which the program interacts with its environment is of importance. Total correctness does not take this interaction into account, so it is not the appropriate correctness criterion for reactive programs. Instead, one should use temporal logic [27] or similar formalisms. Total correctness is, however, the appropriate correctness notion for *parallel algorithms*. These programs differ from sequential algorithms only in that they are executed in parallel, by co-operation of many processes. They are intended to terminate, and only the final results are of interest. The contribution of this paper is a method for deriving this kind of algorithms in a stepwise manner within the refinement calculus.

The derivation of a parallel algorithm for solving linear systems of equations carried out here is a nontrivial example of the stepwise refinement approach. We start from the familiar and completely sequential Gaussian elimination algorithm and derive in a sequence of refinement steps a parallel solution for it. The underlying architecture is assumed to be distributed: the processes have only local memory and limited possibilities to communicate with each other. In Appendix A we give another parallel solution to this problem in which the processes communicate through shared memory.

Many of the refinement steps in this example are described informally. They can, however, be formally proved correct, either directly by using the definition of correct refinement between program statements, or by showing that they are instances of more general rules which can be proved correct once and for all. A number of such rules are given in [2, 4, 24].

Stepwise refinement of parallel programs is also studied by Chandy and Misra [13, 14] with the UNITY approach. UNITY programs are very similar to action systems, although their actions are restricted to conditional and deterministic assignment statements. The approach to refinement is also different from ours: the specification of the program is refined instead of refining the program text as we do here.

Approaches similar to the action system formalism have lately also appeared in several other works [17, 21, 28, 30]. Common to all these approaches is the idea of an event-based description of distributed systems. The roots of this approach can be found in the Petri-net approach, see e.g. [29], and in the works of Dijkstra [15, 16].

The contents of the paper is as follows. The action system formalism is described in Section 2. We also discuss different execution models for action systems and how they are implemented. In Section 3, we present the refinement calculus for statements and actions. Refinement of atomicity in action systems is also discussed in this section.

The main emphasis of the paper is our case study of stepwise refinement where we derive a parallel algorithm for the Gaussian elimination method. The Gaussian

elimination method has two separate phases: triangularization and backsubstitution. Only the first phase is described here. A derivation of the entire algorithm is reported in an accompanying paper [11].

The problem and the sequential Gaussian elimination method is described in Section 4. The parallel algorithm is going to have one process for each column in the coefficient matrix. In Section 5, we describe the refinement steps. Section 5.1 contains an overview of the derivation. In Section 5.2, we show how to transform the triangularization algorithm from a row-based to a column-based sequential algorithm, as a first preparation for parallel execution. In Section 5.3, we transform the triangularization algorithm to a form where the successive pivoting operations become more independent of each other, as a second preparation for parallelization. In Section 5.4, the sequential algorithm is refined to a parallel one, by changing the strictly sequential execution to one where many actions can be executed simultaneously. In Section 5.5, we carry out some minor transformations by which the structure of the action system is simplified. We end with some concluding remarks in Section 6.

2. Action systems

An *action system* $\mathcal{A}$ is a collection of *actions* $\{A_1, \ldots, A_m\}$ on some set of *state variables* $x = \{x_1, \ldots, x_n\}$. Each variable is associated with some domain of values. The set of possible assignments of values to the state variables constitutes the *state space* Σ. Each action A_i is of the form $g_i \rightarrow S_i$ where the *guard* g_i is a boolean condition and the *body* S_i a sequential, possibly nondeterministic statement on the state variables. An *initialization statement* S_0 assigns initial values to the variables x.

An action system describes the state space of a system and the possible actions that can be executed in the system. The way in which the actions are executed depends on the evaluation mechanism that we postulate for the system. We will describe three different kinds of mechanisms, sequential, parallel and distributed execution of action systems.

The behaviour of a *sequential action system* $\mathcal{A}$ is that of the guarded iteration statement

$$S_0;\ \textbf{do}\ A_1 \,\square\, \cdots \,\square\, A_m\ \textbf{od}$$

on the state variables x [15]. The initialization statement is executed first, after which the do-loop is executed, as long as there are actions A_i that are *enabled* (actions whose guards evaluate to *true*). The action system is said to terminate, if any possible execution of the guarded iteration statement terminates. It is said to establish a postcondition R, if it terminates and the final state of any terminating computation satisfies condition R. The guarded iteration statement was introduced by Dijkstra [15], who used his weakest precondition technique to define the semantics of it.

We will use the following syntax to describe an action system

AS :–
var
x_1 **is** T_1; . . . ; x_n **is** T_n
def
p_1 :– D_1; . . . ; p_r :– D_r
begin
S_0;
do $A_1 \,\square \cdots \square\, A_m$ **od**
end

Here *AS* is the name of the action system, x_i is a variable (or list of variables) of type T_i, and each p_i is the definition of a named sequence of statements D_i. We will use indexing quite freely. Thus, for instance, x_i **is** T_i, i: 1..n, stands for x_1 **is** T_1; . . . ; x_n **is** T_n, (i: 1..n: S_i) stands for S_1; S_2; . . . ; S_n and ($\square$ i: 1..m: A_i) stands for $A_1 \,\square\, A_2 \,\square \cdots \square\, A_m$. We also use the dot notation $f.i$ for functional application $f(i)$.

Example. The following is a simple sorting program, described as an action system:

ExchangeSort :–
var
$x.i$ **is** *integer*, i: 1..n
def
Swap.i :–
$x.i, x.(i+1) := x.(i+1), x.i$
begin
(i: 1..n: $x.i := X.\{X \in \mathbb{Z}\}$);
do ($\square$ i: 1..$n-1$: $x.i > x.(i+1) \rightarrow$ *Swap.i*) **od**
end

This program will sort n integers in ascending order. The initial integer value for the variable $x.i$ is chosen nondeterministically in the nondeterministic assignment statement $x.i := X.\{X \in \mathbb{Z}\}$. Each $x.i$ is assigned some value X so that the condition $\{X \in \mathbb{Z}\}$ is satisfied. (We explain this statement in more detail in Section 3.)

We can look upon this program as an action system with an initialization statement and $n-1$ sorting actions. The program obviously terminates in a state where the array x is a permutation of the original array and where $x.i \leqslant x.(i+1)$ for $i = 1, \ldots, n-1$.

2.1. Parallel execution of action systems

We consider here two different ways in which we can execute an action system in parallel, depending on the way in which the system is split up among processes. In a *concurrent action system* the *actions* are partitioned among the processes, while

in a *distributed action system* the *variables* are partitioned among the processes. The first approach leads to a parallel execution model with shared variables used for communication and synchronization, while the second approach results in a distributed execution model where all variables are local and processes communicate by a generalized handshaking mechanism.

2.1.1. Concurrent action system

In a concurrent action system each *action* is assigned to some specific process. A *variable* that is referred to by at least two different actions in two different processes is *shared*, while a variable that is referred to only by actions in one process is *private* to that process.

The actions in the system are executed in parallel with the restriction that all actions are *atomic*: two actions that share a common variable may not execute at the same time. Two actions belonging to the same process may not execute in parallel either, even when they have no variables in common.

The sequential action system is a special case of a parallel action system which we get when all actions are assigned to the same process. Another special case is that each action is assigned to its own process. The system is then executed with maximal parallelism.

Because actions are atomic, a parallel computation can be described as a sequential computation in which the actions of different processes are interleaved. Hence, the set of possible executions of an action system will be the same for a concurrent action system and for a sequential action system. In proving logical properties of action systems, we need therefore consider only the behaviour of the corresponding sequential action system. The two kinds of action systems differ only in the efficiency with which the algorithm is executed and in the way in which the action system is actually implemented. In a real implementation of the parallel action system, the atomicity must be guaranteed by some kind of locking mechanism.

The example sorting program can be interpreted as a concurrent action system by assuming the existence of $n-1$ processes and assigning each of the $n-1$ actions to a process of its own. The variables $x.1, \ldots, x.n$ are shared among the processes (each variable is shared by at most two processes).

2.1.2. Distributed action systems with shared actions

In a distributed action system, each *variable* is assigned to a unique process. An *action* is *shared*, if it refers to variables in two or more processes. If all the variables referred to by an action belong to the same process, then the action is *private* to that process. A shared action is assumed to be executed jointly by all the processes sharing this action. The processes are therefore synchronized for execution of the shared action. Shared actions also provide communication between the processes: a variable of one process may be updated in a way that depends on the values of variables in other processes involved in the shared action.

The actions in a distributed action system are executed in parallel, with the restriction enforced by atomicity. This means that no two actions involving a common process may execute simultaneously. In other words, a process may only participate in one action at a time. Because the variables are partitioned among the processes, this implies that no two actions referring to a common variable may execute in parallel. As in the concurrent action system, this requirement of atomicity implies that the logical behaviour of the distributed action system is the same as that of the corresponding sequential action system.

We can interpret the sorting program as a distributed action system by assigning one element of the array x to each process. We thus need n processes. Each action in the system is then shared between two processes.

2.2. Programming with action systems

The use of action systems permits the design of the logical behaviour of a system to be separated from the issue of how this system is to be implemented. The latter is seen as a design decision that does affect the way in which the action system is built, but is not reflected in the logical behaviour of the system. The decision whether the action system is to be executed in a sequential, concurrent or distributed fashion can be done at a later stage, after the logical behaviour of the action system has been designed. The construction of the program can thus be done within a single unifying framework. A similar separation between logical behaviour and implementation is made in [14].

Given a sequential action system, the concurrent system is determined by how the actions are partitioned among the processes, and the distributed system by how the variables are partitioned among the processes. This determines the degree of parallelism that can be achieved in the system, and, in the distributed case, also the way in which the processes must be connected to each other.

Action systems provide a generalization of the communication mechanisms normally found in programming languages for parallel and distributed programming. Parallel implementation of action systems requires some additional mechanism to enforce atomicity and to schedule the execution of actions. A truly distributed implementation of action systems requires that some constraints are made on these. In [7], it is shown how a class of action systems can be implemented in CSP with output guards [18, 19]. Efficient algorithms to implement action systems on broadcasting networks are presented in [6, 9, 10]. In a forthcoming paper [12] we show how a special class of action systems can be efficiently implemented in occam [20] which does not have output guards.

3. Refinement calculus

In this section we describe the refinement calculus on which our derivation method for parallel algorithms is based. The refinement calculus relies on the weakest

precondition technique of Dijkstra [15] which we describe very briefly below. We then define the refinement relation for statements and actions and give some example proof rules for refinement. Finally, we describe the way in which the atomicity of actions in action systems is refined.

3.1. Weakest preconditions for statements and actions

We restrict ourselves to the language of guarded commands [15] with some extensions. We have two different syntactic categories, *statements* and *actions*, which we define as follows. A *statement* S is defined as

$$\begin{aligned}
S ::= {} & x := x'.Q && \text{(nondeterministic assignment)}\\
| {} & \{Q\} && \text{(assert statement)}\\
| {} & (i: 1..n: S_i) && \text{(sequential composition)}\\
| {} & \textbf{if } (\Box\ i: 1..n: A_i) \textbf{ fi} && \text{(conditional composition)}\\
| {} & \textbf{do } (\Box\ i: 1..n: A_i) \textbf{ od} && \text{(iterative composition)}\\
| {} & [\textbf{var } x_i: T_i, i: 1..n; S] && \text{(block with local variables).}
\end{aligned}$$

Here $A_1, \ldots, A_n$ are actions, x and x' are (lists of) variables and Q is a predicate. The *nondeterministic assignment statement* $x := x'.Q$ [2] assigns to the variables x some values x' that make the condition Q true. The statement aborts, if this is not possible. Here x is a list of distinct variables and x' is a list of fresh variables local to this statement. The *assert statement* $\{Q\}$ acts as *skip*, if the condition Q holds in the initial state. If the condition Q does not hold in the initial state, then the effect is the same as *abort*. Hence, $\{true\} = skip$ and $\{false\} = abort$. The other statements have their usual meanings. The (*multiple*) *assignment statement*, $x := e$, is a special case of the nondeterministic assignment statement. It is defined as

$$(x := e) = (x := x'.(x' = e)),$$

for some set of fresh variables x'. In the block construct, the indication of scope of local variables is normally omitted. In a sequential construction we assume that the scope is as large as possible, i.e.,

$$\textbf{var } x; S_1; \ldots; S_n = [\textbf{var } x; S_1; \ldots; S_n].$$

An *action* (or guarded command) A is of the form

$$A ::= g \to S$$

where g is a boolean condition (the *guard*) and S is a statement (the *body*).

An action system $\mathscr{A}$ is simply a statement of the form

$$[\textbf{var } x_i: T_i, i: 1..m; S_0; \textbf{do } (\Box\ i: 1..n: A_i) \textbf{ od}]$$

where $A_1, \ldots, A_n$ are actions. The variables $x_1, \ldots, x_m$ are local to the action system. The other variables referenced in the system are global.

The *weakest precondition*, $\mathrm{wp}(S, R)$, for any statement S and predicate R is a predicate that describes the set of all states such that execution of S begun in any

one of them is guaranteed to terminate in a state satisfying R. The weakest precondition for the assert statement is

$$\mathrm{wp}(\{B\}, R) = B \wedge R.$$

The weakest precondition for the nondeterministic assignment statement is

$$\mathrm{wp}(x := x'.Q, R) = \exists x'.Q \wedge \forall x'.(Q \Rightarrow R[x'/x]).$$

The weakest precondition for blocks is

$$\mathrm{wp}([\textbf{var } x; S], R) = \forall x.\mathrm{wp}(S, R).$$

(We assume that nested declarations of the same variable are disallowed.) The weakest preconditions of the other statements are computed using the familiar rules in [15], with the adaptions needed to permit unbounded nondeterminism [4, 25].

The weakest precondition for an action $A = g \rightarrow S$ to establish postcondition R is

$$\mathrm{wp}(g \rightarrow S, R) = (g \Rightarrow \mathrm{wp}(S, R)).$$

The weakest precondition for a finite sequence $c = (i: 1..k: A_i)$ of actions A_i, $i = 1, \ldots, k$, to establish postcondition R is calculated using the rule of sequential composition for weakest preconditions:

$$\mathrm{wp}((i: 1..k: A_i), R) = \mathrm{wp}(A_1, \mathrm{wp}((i: 2..k: A_i), R)), \quad k > 1.$$

An empty sequence acts as a *skip* statement.

Observe that the law of excluded miracle, $\mathrm{wp}(S, \mathit{false}) = \mathit{false}$, does not necessarily hold for actions, [5, 22, 25]. As an example, we have that $\mathrm{wp}(\mathit{false} \rightarrow \mathit{skip}, \mathit{false}) = \mathit{true}$. Also, the continuity property need not hold, as the nondeterminism of the nondeterministic assignment statement may be unbounded. The other healthiness properties of [15] are valid.

3.2. *Refinement of statements and actions*

Let S and S' be sequential statements. Statement S is said to be *refined* by the statement S', denoted $S \leqslant S'$, if

$$\mathrm{wp}(S, R) \Rightarrow \mathrm{wp}(S', R)$$

for every postcondition R. Refinement captures the notion of statement S' preserving the correctness of statement S. Hence, if S is totally correct with respect to a given P and Q, and $S \leqslant S'$, then S' will also be totally correct with respect to P and Q.

We say that the statements S and S' are (*refinement*) *equivalent*, denoted $S \equiv S'$, if $S \leqslant S'$ and $S' \leqslant S$.

The refinement relation is reflexive and transitive, i.e., it is a preorder. It is also monotonic, i.e.,

$$(T \leqslant T') \Rightarrow (S(T) \leqslant S(T'))$$

for any sequential statement $S(T)$ that contains T as a substatement. If the semantics

of a statement is defined by its weakest precondition predicate transformer, then refinement is also antisymmetric, i.e., it is a partial order. Refinement equivalence is then the identity relation.

The monotonicity of the statement constructors with respect to the refinement relation means that if $S_i \leq S_i'$, for $i = 1, \ldots, n$, then

$$(i\colon 1..n\colon S_i) \leq (i\colon 1..n\colon S_i'),$$

$$\textbf{if } (\Box\ i\colon 1..n\colon g_i \to S_i)\ \textbf{fi} \leq \textbf{if } (\Box\ i\colon 1..n\colon g_i \to S_i')\ \textbf{fi},$$

$$\textbf{do } (\Box\ i\colon 1..n\colon g_i \to S_i)\ \textbf{od} \leq \textbf{do } (\Box\ i\colon 1..n\colon g_i \to S_i')\ \textbf{od} \quad \text{and}$$

$$[\textbf{var } x;\ S_1] \leq [\textbf{var } x;\ S_1'].$$

The refinement relation provides a formalization of the stepwise refinement method for program construction. One starts with an initial program S_0, and constructs a sequence of successive refinements of this, $S_0 \leq S_1 \leq \cdots \leq S_{n-1} \leq S_n$. By transitivity, the last version S_n will then be a refinement of the original program S_0. An individual refinement step may consist of replacing some substatement T of $S_i(T)$ by its refinement T'. The resulting statement $S_{i+1} = S_i(T')$ will then be a refinement of S_i. The refinement relation is studied in more detail in [2, 4, 23, 25].

3.2.1. Context-dependent replacements

The monotonicity of refinement allows us to replace a substatement T in $S(T)$ with some other statement T' if $T \leq T'$ holds. It is, however, possible that the replacement of T by T' would preserve correctness in the specific context $S(\cdot)$, i.e., $S(T) \leq S(T')$, although the replacement would not be correct in every possible context. In this case $T \leq T'$ does not hold, so the replacement is not justified by monotonicity alone. This kind of *context-dependent replacements* are very important in practice. They can be handled by the following technique.

Assume that we prove

(1) $S(T) \leq S(\{Q\};\ T)$,
(2) $\{Q\};\ T \leq T'$.

By monotonicity and transitivity we then have that $S(T) \leq S(T')$.

The first step is called *context introduction*, while the second step is called *refinement in context*. Context introduction is needed to show what assumptions can be made about the context in which the replacement is to be made. Refinement in context is a weaker requirement than the usual refinement, in that it requires that the statement S is refined by the statement S' only for those initial states in which the condition $\{Q\}$ holds. This technique is described in more detail in [2, 4], together with rules for context introduction in sequential statements.

An important special case of context introduction is the use of program invariants. More precisely, if we have shown that for a given iterative composition

$$\{P\};\ \textbf{do } (\Box\ i\colon 1..n\colon g_i \to S_i)\ \textbf{od}$$

the condition P is an invariant of the loop, then the iterative composition

$$\{P\};\ \textbf{do}\ (\Box\ i{:}\ 1..n{:}\ g_i \wedge P \rightarrow \{P\};\ S_i)\ \textbf{od}$$

is refinement equivalent to the original iterative composition. This allows us to introduce information about the context in which the action is used into the action itself. This is a very useful technique when attempting to do refinement of action systems in practice. Most refinement steps are only valid in the specific context in which they are performed, so one has to introduce context information to be able to carry out the refinement step.

Note that we are always permitted to remove any context assertion, i.e.,

$$S(\{Q\};\ T) \leqslant S(T)$$

is always valid (because $\{Q\};\ T \leqslant T$ is always valid).

3.2.2. Refinement of actions

Refinement between actions is defined in the same way as refinement between statements, i.e., an action A is *refined by* an action A', $A \leqslant A'$, if

$$\mathrm{wp}(A, R) \Rightarrow \mathrm{wp}(A', R)$$

for any postcondition R. Action $A = g \rightarrow S$ is thus refined by action $A' = g' \rightarrow S'$ if

$$[g \Rightarrow \mathrm{wp}(S, R)] \Rightarrow [g' \Rightarrow \mathrm{wp}(S', R)],$$

for any postcondition R. This is equivalent to the following two conditions:

(1) $g' \Rightarrow g$, i.e., A is always enabled when A' is enabled,
(2) $\{g'\};\ S \leqslant S'$, i.e., the body of A is refined by the body of A' whenever A' is enabled.

Observe that even if refinement of statements is monotonic, as stated above, action systems are *not* necessarily monotonic with respect to refinement of actions, i.e., $(g_i \rightarrow S_i) \leqslant (g_i' \rightarrow S_i') \Rightarrow (S_0;\ \textbf{do}\ \ldots \Box\ g_i \rightarrow S_i \ldots\ \textbf{od}) \leqslant (S_0;\ \textbf{do}\ \ldots \Box\ g_i' \rightarrow S_i' \ldots\ \textbf{od})$ need not hold (unless $g_i = g_i'$).

3.3. Some example refinement rules

The refinement calculus presented so far can be used to derive a number of useful program transformation rules. Here we derive one such rule as an example. We also describe a method for introducing and removing local variables during a program derivation. These examples are chosen with our program derivation in mind.

3.3.1. Changing variables

Let S be a statement that does not contain any occurrence of x. Then

$$S \equiv [\textbf{var}\ x;\ S[x := h_1/\mathit{skip}^1, \ldots, x := h_n/\mathit{skip}^n]], \qquad (*)$$

where $\mathit{skip}^1, \ldots, \mathit{skip}^n$ denote different occurrences of the *skip* statement in S. In

other words, we are free to introduce new local variables and assignments to them in any statement. This refinement equivalence can be proved correct in the weakest precondition calculus by structural induction.

We use (*) to give a method for changing local variables in a statement. Assume that we have initially a statement [**var** x; S] and we want to replace the variables x with some other variables y, changing S to S' accordingly, such that

$$[\textbf{var } x;\ S] \leq [\textbf{var } y;\ S'].$$

We can achieve this by the following sequence of steps:

Step 1. *Introduce new variables* y. Let $S_1 = S[y_1 := e_1/\mathit{skip}^1, \ldots, y_n := e_n/\mathit{skip}^n]$. Applying (*) and monotonicity of refinement, we have

$$[\textbf{var } x;\ S] \equiv [\textbf{var } x;\ [\textbf{var } y;\ S_1]].$$

Step 2. *Introduce context assertion relating* x *and* y. Let $Q_1, \ldots, Q_k$ be assertions on x and y, and assume that context introduction gives us

$$S_1 \leq S_1[\{Q_1\};\ T_1/T_1, \ldots, \{Q_k\};\ T_k/T_k],$$

where $T_1, \ldots, T_k$ are substatements of S_1 that refer to the variables in x.

Step 3. *Replace statement on* x *with statement on* y. Let $T'_1, \ldots, T'_k$ be statements that do not refer to variables in x, and assume that we can show that

$$\begin{aligned} &\{Q_1\};\ T_1 \leq T'_1 \\ &\quad\vdots \\ &\{Q_k\};\ T_k \leq T'_k. \end{aligned}$$

By transitivity of refinement, we now have

$$S_1 \leq S_1[T'_1/T_1, \ldots, T'_k/T_k] = S_2.$$

Step 4. *Remove old variables* x. Assume that all the remaining occurrences of x are now in assignments to variables in x, i.e.,

$$S_2 = S'_2[x := h_1/\mathit{skip}^1, \ldots, x := h_m/\mathit{skip}^m],$$

where S'_2 does not contain any occurrences of x.

Applying (*) again then gives us

$$\begin{aligned} [\textbf{var } x;\ [\textbf{var } y;\ S_1]] &\leq [\textbf{var } x;\ [\textbf{var } y;\ S_2]] \\ &\equiv [\textbf{var } y;\ [\textbf{var } x;\ S_2]] \\ &\equiv [\textbf{var } y;\ [\textbf{var } x;\ S'_2[x := h/\mathit{skip}^1, \ldots, x := h_m/\mathit{skip}^m]]] \\ &\equiv [\textbf{var } y;\ S'_2]. \end{aligned}$$

Transitivity of refinement then gives us the desired result

$$[\textbf{var } x;\ S] \leq [\textbf{var } y;\ S'_2].$$

3.3.2. Refining a sequence with a loop

We have the following refinement rule: If

(1) $\{g_i\}; S_i \leq S_i; \{g_{i+1}\}, \quad i = 1, \ldots, n,$

(2) $g_i \Rightarrow (\forall j: j \in \{1, \ldots, n+1\} \wedge j \neq i: \neg g_j), \quad i = 1, \ldots, n+1,$

then

$$\{g_1\}; (i: 1..n: S_i) \equiv \{g_1\}; \textbf{do } (\Box\ i: 1..n: g_i \rightarrow S_i) \textbf{ od}$$

We prove this as follows. Let $gg = \bigvee_{i=1}^{n} g_i$. Then

$\{g_1\}$; **do** $(\Box\ i: 1..n: g_i \rightarrow S_i)$ **od**

$\equiv$ [unfold loop one step]

$\{g_1\}$; **if** $(\Box\ i: 1..n: g_i \rightarrow S_i;$ **do** $(\Box\ i: 1..n: g_i \rightarrow S_i)$ **od**)

$\quad \Box \neg gg \rightarrow skip$

$\quad$ **fi**

$\equiv$ [by (2)]

$\{g_1\}; S_1;$ **do** $(\Box\ i: 1..n: g_i \rightarrow S_i)$ **od**

$\equiv$ [by (1)]

$\{g_1\}; S_1; \{g_2\};$ **do** $(\Box\ i: 1..n: g_i \rightarrow S_i)$ **od**

$\equiv$ [unfold loop one step]

$\vdots$

$\equiv$ [unfold loop one step]

$\{g_1\}; S_1; \{g_2\}; S_2; \ldots; \{g_n\}; S_n;$ **do** $(\Box\ i: 1..n: g_i \rightarrow S_i)$ **od**

$\equiv$ [by (1)]

$\{g_1\}; S_1; \{g_2\}; S_2; \ldots; \{g_n\}; S_n; \{g_{n+1}\};$ **do** $(\Box\ i: 1..n: g_i \rightarrow S_i)$ **od**

$\equiv$ [by (2), $g_{n+1} \Rightarrow \neg gg$]

$\{g_1\}; S_1; \{g_2\}; S_2; \ldots; \{g_n\}; S_n; \{g_{n+1}\}; skip$

$\equiv$

$\{g_1\}; S_1; S_2; \ldots; S_n.$

The last step requires some explanation. By (1), we have that $(\{g_1\}; S_1; S_2; \ldots; S_n) \leq (\{g_1\}; S_1; \{g_2\}; S_2; \ldots; S_n) \leq \cdots \leq (\{g_1\}; S_1; \{g_2\}; S_2; \ldots; \{g_n\}; S_n; \{g_{n+1}\})$. Refinement in the other direction follows directly, because we may always remove context assertions.

3.4. Refining atomicity in action systems

The actions in an action system are treated as atomic when the action system is executed in parallel. Hence the coarseness of the actions determines the degree of parallelism that is achieved during execution. To increase the parallelism, one therefore tries to split up larger actions into a number of smaller actions.

Consider an action system

$$\mathcal{B} = T_0; \textbf{do } A \Box B_1 \Box \cdots \Box B_k \textbf{ od}.$$

Assume that we want to split up the action $A = g \to S$ into a number of smaller actions, such that the resulting system $\mathscr{B}'$ is a refinement of the original action system.

This refinement can be done in two steps. First we give an action system

$$\mathscr{A} = S_0;\ \textbf{do}\ A_1 \,\square\, \cdots \,\square\, A_m\ \textbf{od}$$

that is a refinement of the statement S in context g, i.e., $(\{g\};\ S) \leqslant \mathscr{A}$. By monotonicity of refinement, we then have that

$$\mathscr{B} \leqslant \mathscr{B}[\mathscr{A}/S]$$

where

$$\mathscr{B}[\mathscr{A}/S] = T_0;\ \textbf{do}\ g \to \mathscr{A} \,\square\, B_1 \,\square\, \cdots \,\square\, B_k\ \textbf{od}.$$

This determines the way in which we want to split up the action A. However, statement $\mathscr{A}$ is still executed as one big action, i.e., atomicity has not been refined.

Our next task is to show that the whole action system $\mathscr{A}$ need not be executed atomically, but that it is sufficient that the individual actions in $\mathscr{A}$ are executed atomically. In other words, we must show that

$$\mathscr{B}[\mathscr{A}/S] \leqslant \mathscr{B}',$$

where $\mathscr{B}'$ is the action system

$$T_0;\ \textbf{do}\ A_0 \,\square\, \cdots \,\square\, A_m \,\square\, B_1 \,\square\, \cdots \,\square\, B_k\ \textbf{od}.$$

Here $A_0 = g \to S_0$ is an additional action that initializes execution of the actions in $\mathscr{A}$.

Before stating the conditions under which this is a correct refinement, let us define some additional concepts.

An action B is said to *commute left* with an action A, if $(B;\ A) \leqslant (A;\ B)$ and to *commute right* with A if $(A;\ B) \leqslant (B;\ A)$. (Note the reversal of direction: we think of $(A;\ B)$ as being reduced to $(B;\ A)$, by moving B left over A.) Let $A = g \to S$ and $A' = g' \to S'$ be two actions. We say that A' is *not disabled* by A, if $g' \Rightarrow \mathrm{wp}(A, g')$. We say that A' is *not enabled* by A, if $\neg g' \Rightarrow \mathrm{wp}(A, \neg g')$. We say that an action A *cannot be followed* by an action A', if $g \Rightarrow \mathrm{wp}(S, \neg g')$.

Commutativity can be proved by using the following result: $(A;\ A') \leqslant (A';\ A)$, if

(1) A' is not disabled by A,
(2) A is not enabled by A',
(3) $(\{g \wedge g'\};\ S;\ S') \leqslant (S';\ S)$.

3.4.1. Atomicity refinement

Let us call the actions $B_1, \ldots, B_k$ above the *old actions* and the actions $A_0, \ldots, A_m$ the *new actions*. The refinement of atomicity $\mathscr{B}[\mathscr{A}/S] \leqslant \mathscr{B}'$ is correct, if the following conditions are satisfied [5]:

(1) An old action (or initialization) cannot enable or disable any new action except A_0.

(2) Action A_0 is never enabled when one of the actions $A_1, \ldots, A_m$ is enabled.
(3) Let $\{B'_1, \ldots, B'_r\}$ be those old actions that can be enabled when one of the actions $A_0, \ldots, A_m$ is enabled. There is a partitioning of these old actions into two disjoint sets, the *left movers* and the *right movers*, such that the following holds:
  (3.1) For each left mover B and each new action A_i, either A_i cannot be followed by B or B commutes left with A_i.
  (3.2) For each right mover B and each new action or left mover C, either B cannot be followed by C, or B commutes right with C.
  (3.3) The action system **do** $(\Box\, i{:}\, 1..k{:}\, R_i)$ **od** consisting of the right movers $R_1, \ldots, R_k$ terminates when one of the actions $A_0, \ldots, A_m$ is enabled.

These are thus the conditions under which the action system $\mathscr{A}$ can be executed non-atomically in the context of the action system $\mathscr{B}$.

4. Case study: Gaussian elimination

As a case study, we will show how to derive a parallel algorithm for solving linear equation systems. We start from a completely sequential and traditional algorithm, based on Gaussian elimination, and in a sequence of refinement steps derive a highly parallel version of this algorithm. We will not give the complete derivation here, to avoid overburdening the presentation. The complete derivation is described in [11].

4.1. Problem statement

We are to solve an equation system of the form

$$\begin{aligned} &a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n = b_1 \\ &a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n = b_2 \\ &\quad\vdots \\ &a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n = b_n \end{aligned}$$

by Gaussian elimination. This equation system is of the form $Ax = b$ where A is an $n \times n$ matrix and x and b are $n \times 1$ matrices,

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & & & \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix}, \qquad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

A is assumed to be nonsingular, so there will be a unique solution to the equation system.

Gaussian elimination makes use of the following properties of matrices:

(1) When a row (or column) of a nonsingular matrix is multiplied by a constant, the resulting matrix is nonsingular.
(2) When a row (column) is added to another row (column) in a nonsingular matrix, the resulting matrix is nonsingular.
(3) When two rows (columns) are exchanged in a nonsingular matrix, the resulting matrix is nonsingular.

These properties actually state that the determinant of the coefficient matrix is not changed by the above row and column operations.

The algorithm works on the matrix $Ab = [A, b]$ where b is added as the $(n+1)$st column of the matrix A. It is based on the fact that performing any of the above-mentioned manipulations on the rows (or columns) of the matrix Ab does not change the solution to the equation system. In other words, if we derive the matrix $Ab' = [A', b']$ from $Ab = [A, b]$ by using these manipulations, then A' will be nonsingular if A is nonsingular, and x will be a solution to the equation system $Ax = b$ if and only if x is a solution to the equation system $A'x = b'$.

Gaussian elimination has two separate phases, *triangularization* and *backsubstitution*. In triangularization, the matrix $[A, b]$ is transformed using the above mentioned row operations into a matrix $[A', b']$, where A' is an upper triangular matrix, i.e., all elements in A' below the diagonal are zeros. The equation system is thus of the form

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= b_1 \\ a_{2,2}x_2 + \cdots + a_{2,n}x_n &= b_2 \\ &\vdots \\ a_{n,n}x_n &= b_n. \end{aligned}$$

The system is then solved by backsubstitution. First, the last variable x_n is solved directly, from the last row. Then the variable x_{n-1} is solved from the previous row, using the computed value of x_n, and so on, until the values of all variables $x_1, \ldots, x_n$ have been computed. In this paper we consider only the triangularization phase. The derivation of the complete Gaussian elimination algorithm is described in [11].

The triangularization program works on the following global data structure

```
const
   n is integer
var
   Ab.j.k is real, j,k: 1..n, 1..n+1.
```

(We will follow the convention that j ranges over rows and k over columns.)

The preconditions for the triangularization phase are:

(1) $Ab = [A_0, b_0]$.
(2) A_0 is nonsingular, i.e., $\det(A_0) \neq 0$.

The postconditions for the triangularization phase are:

(1) $Ab = [A, b]$.
(2) A is nonsingular, i.e., $\det(A) \neq 0$.
(3) $Ay = b$ has the same solutions as $A_0 y = b_0$, i.e., $\forall y$: $(Ay = b \Leftrightarrow A_0 y = b_0)$.
(4) A is upper triangular, i.e., $\forall i$: $1 \leq i \leq n$: $\forall j$: $i < j \leq n$: $A[j, i] = 0$.
(5) The diagonal of A contains no zeros, i.e., $\forall i$: $1 \leq i \leq n$: $A[i, i] \neq 0$.

Observe that postcondition (5) follows from postconditions (2) and (4). We will, however, use the above postconditions for clarity.

4.2. *Triangularization algorithm*

We start with a completely sequential version of the triangularization. The basic manipulation that we do is *pivoting*: Given the rows

$$r_i = a_{i,i}x_i + a_{i,i+1}x_{i+1} + \cdots + a_{i,n}x_n = b_i,$$

$$r_j = a_{j,i}x_i + a_{j,i+1}x_{i+1} + \cdots + a_{j,n}x_n = b_j,$$

we can eliminate the variable x_i from row r_j by the operation

$$r_j := r_j - p_{j,i} * r_i$$

where

$$p_{j,i} = a_{j,i} / a_{i,i}.$$

This operation is permitted when $a_{i,i} \neq 0$.

The algorithm first eliminates the variable x_1 from rows $r_2, \ldots, r_n$ by pivoting, then the variable x_2 is eliminated from rows $r_3, \ldots, r_n$ and so on. In the last step, the variable x_{n-1} is eliminated from the row r_n. The result is an upper triangular matrix. To avoid the problem where the pivot element $a_{i,i}$ of the pivot row r_i is zero, the rows below r_i are first scanned to find the row r_j which has the largest absolute value of the coefficient $a_{j,i}$. The rows r_j and r_i are then exchanged before the pivoting is done. This is known as *partial pivoting*. If the matrix is nonsingular, then there always exists a row r_i which has a nonzero coefficient for x_i. (The largest coefficient is chosen to minimize the accumulated effect of rounding errors.)

4.2.1. *The triangularization algorithm*

The matrix Ab is transformed into upper triangular form by the following action system

Triangularize :–
def
FindMaxRow.i :–
maxrow := *i*;
(*j*: *i*+1..*n*: **if** |*Ab.j.i*| > |*Ab.maxrow.i*| **then** *maxrow* := *j* **fi**)

```
  Exchange.i :-
    (k: i..n+1: Ab.i.k, Ab.maxrow.k := Ab.maxrow.k, Ab.i.k)
  EliminateRow.i :-
    var p is real;
    (j: i+1..n: p := Ab.j.i/Ab.i.i;
                (k: i..n+1: Ab.j.k := Ab.j.k - p * Ab.i.k))
  Pivot.i :-
    var maxrow is integer;
    FindMaxRow.i;
    Exchange.i;
    EliminateRow.i
begin
  (i: 1..n-1: Pivot.i)
end
```

We write **if** b **then** S **fi** for **if** $b \to S \,\square\, \neg b \to skip$ **fi**. For simplicity, we also assume that $(x, x := e, e) \equiv (x := e)$ (needed in *Exchange.i*). We will follow the convention that i ranges over the pivot steps.

The correctness of this algorithm with respect to the pre- and postconditions given above can be shown by the usual invariant assertion method. The main invariant of this algorithm is shown pictorically in Fig. 1.

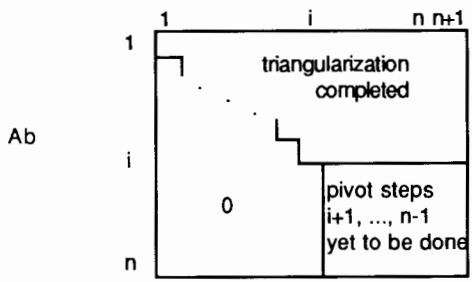

Fig. 1. Pivoting step i.

5. Refinement steps

In this section we show how the above sequential triangularization algorithm can be transformed into a version where the different pivoting steps can proceed in parallel. Our aim is to find an algorithm where each processor computes the values of one column in the coefficient matrix.

We derive below a distributed solution to the problem, i.e., each processor is assumed to have local memory only. We further assume that the processors are connected in an array structure and that each processor is capable of communicating

with its two neighbouring processors, with the restriction that communication takes place with only one processor at a time. In terms of action systems this means that the variables of the action system are partitioned among the processes of the target architecture (one process is assigned to each processor) and that each shared action can refer to the variables of at most two adjacent processes. In Appendix A we give a parallel solution to the problem when we assume that the communication takes place via shared memory. In this case an action can refer to any variables of the action system.

Observe that during the early stages of the derivation procedure we do not respect the process boundaries. Our knowledge of the structure of the target system will, however, guide our transformations.

5.1. Overview of the refinement steps

First, we briefly describe the different refinement steps taken, in order to give the reader an overview of the derivation process. The detailed derivation starts in Section 5.2.

Refinement step 1: Column-based triangularization

The purpose of refinement step 1 is to transform the row-based initial algorithm to a column-based version to reflect the intended partition of the coefficient matrix. The transformation is based on the observation that in pivoting step i all the needed multipliers $p_{j,i}$ for $j = i+1, \ldots, n-1$ can be computed before the coefficient matrix is updated (see Section 4). Thereafter the columns in the matrix can be updated one at a time.

The result of this step is an algorithm where there are two phases for each pivoting step i, $i = 1, \ldots, n-1$. In the first phase we compute the multipliers $p.j$ for $j = i+1, \ldots, n$ and update the matrix elements $Ab.j.i$ for $j = i+1, \ldots, n$. In the second phase the coefficient matrix $Ab.j.k$ is updated for $j = i+1, \ldots, n$, $k = i+1, \ldots, n+1$.

Refinement step 2: Making pivot operations more independent

This refinement step is carried out in order to make it possible for several pivoting steps to proceed in parallel in the column-based version of the triangularization algorithm.

Assume that the pivoting steps l and k, $l \neq k$, in the previous algorithm are performed concurrently. We immediately notice that these two pivoting steps utilize the same set of multipliers $p.j$ and the same *maxrow* variable. As there exists a unique set of multipliers for each pivoting step, we store them separately for each pivoting step in the array $p.i.j$, $i = 1, \ldots, n-1$, $j = 2, \ldots, n$. Also the row number of the unique *maxrow* value in each pivot element computation is stored per pivoting step.

As a result, the values needed to perform a pivoting step are now available even when the steps are done in parallel. The matrix updating must, however, be executed in a strictly sequential manner: the computation concerning column j for pivoting

step k must preceed the computation done for the same column in step l, $l > k$ where $j > l$. This sequencing is accomplished by keeping track of the number of performed pivoting steps. The information is stored in the variable *iter.i*, $i = 1, \ldots, n+1$ per column.

After the first phase of pivoting step i (and before pivoting step $i+1$) the value of *iter.i* equals $i+1$, denoting that pivoting is completed for column i. The values of *iter.l*, for $l = i+1, \ldots, n+1$, equal $i-1$. The second phase of step i increments the values *iter.l*, for $l = i+1, \ldots, n+1$, by one. We thus have the situation described in Fig. 2 after the completion of pivoting step i and before pivoting step $i+1$.

Refinement step 3: Distributed triangularization

The main transformation in refinement step 3 is the refinement of atomicity performed in the triangularization algorithm. We need, however, some preliminary work.

In this step we take into account the restriction that the processor controlling column k is capable of communicating with processors $k-1$ and $k+1$ only. Assume that all the variables are partitioned among the processes of the system. In addition to the coefficient matrix, each column process i is also responsible for maintaining the pivoting information (the multipliers and the maxrow number) concerning pivoting step i.

Let us consider pivoting step i and column process k for some $k > i$. The pivoting information for this step is stored in process i. When updating the coefficient matrix in column k we must perform a shared action which refers to variables situated in the two column processes i and k. However, a shared action referring to variables in process k can in addition to these refer to variables in the processes $k-1$ and $k+1$ only. We thus need the pivoting information to flow through the column processes. This effect is achieved by adding some new variables and assignments to these. We also transform the matrix updating so that the new variables are used in computations.

After some minor transformations our triangularization algorithm can be understood as an action system with an initialization statement and $n-1$ huge pivot actions. The atomic pivot action i computes the pivoting information, updates column i and thereafter updates the coefficient matrix columnwise from column $i+1$ to $n+1$. The enabledness of this action depends on the values of the iteration counters so that the action is enabled only when the situation in Fig. 2 holds for the previous pivoting step $i-1$.

The following major transformation is the atomicity refinement of the pivot actions. We observe that pivot action i can be refined into $n-i+1$ smaller actions: one action to compute the pivoting information and to update column i, plus $n-i$ actions each concerning the update of one column in the coefficient matrix. The first action is enabled when the situation in Fig. 3 holds for columns $i-1$, i and $i+1$. A matrix updating in column j for $j > i$ for pivoting step i is enabled when we have the situation in Fig. 4 holding for the columns $j-1$, j and $j+1$.

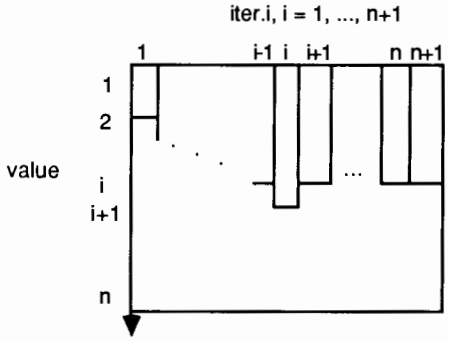

Fig. 2. The array *iter.i*, $i = 1, \ldots, n+1$.

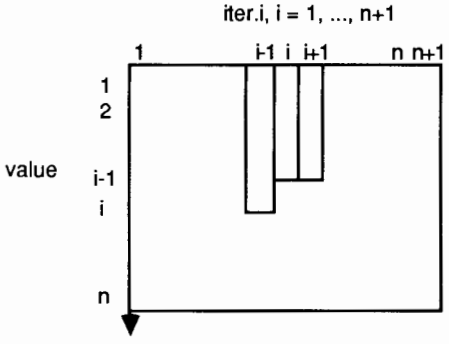

Fig. 3. Initiating pivoting step *i*.

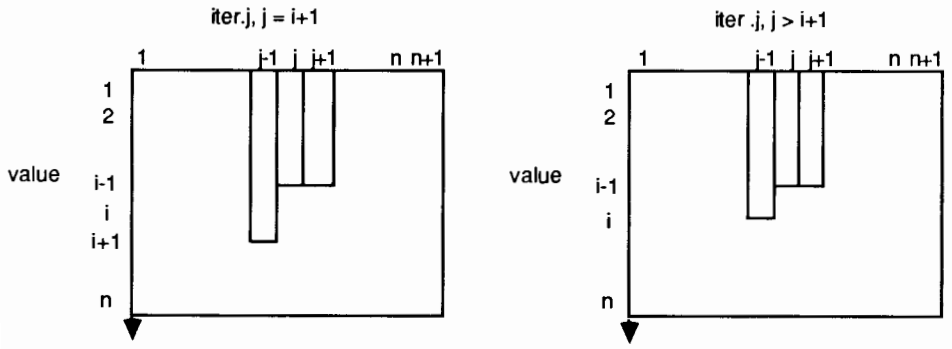

Fig. 4. Matrix updating in pivoting step *i*.

Due to the way we pass the pivoting information among the column processes, the pivot element computation and matrix updating must be performed in the strictly sequential order $i, i+1, \ldots, n+1$ for each pivoting step i. This same information wave places also an other constraint on action activation. Namely, two matrix updatings belonging to different pivoting steps are not allowed to pass each other.

The two waves must be as a matter of fact kept at least one column apart in order not to destroy the pivoting information stored in a column process. This explains the need to check the iteration counters of three columns in each action guard.

Refinement step 4: Merging actions

At this point we have derived a distributed action system for our example problem where each action requires the co-operation of three column processes. Values of variables residing in three column processes are needed in order to evaluate the action guards, but the action bodies refer to variables in two processes only. A fourth refinement step is needed to remove the third process from the action guards.

The main transformation of this step is based on the observation that, in addition to the pivoting information, we can pass information about the values of the iteration counters between the involved column processes. This is done by introducing a new variable *equal.k* in each column process k to denote whether the iteration counters for columns k and $k+1$ are equal or not. In this way we only need to inspect the values of the variables in two neighbouring columns when evaluating action guards.

5.2. Refinement step 1: Column-based triangularization

Our intention is to construct a multiprocessor algorithm for the triangularization phase where each column of the matrix is allocated to one processor. Hence, we need to change the algorithm so that it reflects this partitioning. (We use braces and reference numbers to identify the statements that are refined.)

5.2.1. Column-oriented operations

Let us look at the *Pivot.i* operation more closely. The operation is described in terms of operations on rows. We will change it into a column-based version. Consider first the step *EliminateRow.i*. We apply the method for changing variables that we described in Section 3.3. We start by replacing the variable p with a different variable $p.j$ for each row j:

EliminateRow.i

$\equiv$ [by definition]

var p **is** *real*;
$(j: i+1..n: p := Ab.j.i / Ab.i.i;$
$\quad (k: i..n+1: Ab.j.k := Ab.j.k - p * Ab.i.k))$

$\leqslant$ [add new local variables $p.j$]

var
$\quad p$ **is** *real*;
$\quad p.j$ **is** *real*, $j: i+1..n$;
$(j: i+1..n: p := Ab.j.i / Ab.i.i;\ p.j := Ab.j.i / Ab.i.i;$
$\quad (k: i..n+1: Ab.j.k := Ab.j.k - p * Ab.i.k))$

$\leqslant$ [add context assertions relating p and $p.j$]

var
p **is** *real*;
$p.j$ **is** *real*, j: $i+1..n$;
$(j: i+1..n: p := Ab.j.i/Ab.i.i;\ p.j := Ab.j.i/Ab.i.i;\ \{p.j = p\};$
$(k: i..n+1: Ab.j.k := Ab.j.k - p * Ab.i.k))$

$\leq$ [change statement to use $p.j$ instead of p]

var
p **is** *real*;
$p.j$ **is** *real*, j: $i+1..n$;
$(j: i+1..n: p := Ab.j.i/Ab.i.i;\ p.j := Ab.j.i/Ab.i.i;\ \{p.j = p\};$
$(k: i..n+1: Ab.j.k := Ab.j.k - p.j * Ab.i.k))$

$\leq$ [remove declaration of p and statements with p]

var $p.j$ **is** *real*, j: $i+1..n$;
$(j: i+1..n: p.j := Ab.j.i/Ab.i.i;$
$(k: i..n+1: Ab.j.k := Ab.j.k - p.j * Ab.i.k))$ (1)

$\equiv$ [new version]
EliminateRow.i, *version* 2

We can split up (1) into two separate sequences, because there is a separate variable $p.j$ for each row j:

(1)
$\equiv$
$(j: i+1..n: p.j := Ab.j.i/Ab.i.i;$
$(k: i..n+1: Ab.j.k := Ab.j.k - p.j * Ab.i.k))$

$\leq$ [$p.j$ can be precomputed]

$(j: i+1..n: p.j := Ab.j.i/Ab.i.i);$
$(j: i+1..n: (k: i..n+1: Ab.j.k := Ab.j.k - p.j * Ab.i.k))$ (2)

Our next step is to change the order of evaluation in the nested sequence (2) so that it proceeds by column first rather than row first order:

(2)
$\equiv$
$(j: i+1..n: (k: i..n+1: Ab.j.k := Ab.j.k - p.j * Ab.i.k))$

$\leq$ [computed elements do not depend on each other, assignments are pairwise commutative]
$(k: i..n+1: (j: i+1..n: Ab.j.k := Ab.j.k - p.j * Ab.i.k))$

$\leq$ [unfold sequence one step]
$(j: i+1..n: Ab.j.i := Ab.j.i - p.j * Ab.i.i);$
$(k: i+1..n+1: (j: i+1..n: Ab.j.k := Ab.j.k - p.j * Ab.i.k))$

Monotonicity of refinement now allows us to derive the following version of *EliminateRow.i*:

$$\left.\begin{array}{l} (j\colon i+1..n\colon p.j := Ab.j.i / Ab.i.i); \\ (j\colon i+1..n\colon Ab.j.i := Ab.j.i - p.j * Ab.i.i); \\ (k\colon i+1..n+1\colon (j\colon i+1..n\colon Ab.j.k := Ab.j.k - p.j * Ab.i.k)) \end{array}\right\} \quad (3)$$

EliminateRow.i, version 3 :–
var *p.j* **is** *real*, *j*: $i+1..n$;

A further refinement of (3) gives us:

(3)

$\equiv$

$(j\colon i+1..n\colon p.j := Ab.j.i / Ab.i.i);$
$(j\colon i+1..n\colon Ab.j.i := Ab.j.i - p.j * Ab.i.i)$

$\leqslant$ [fuse the sequences on *j*, assignments are pairwise commutative]

$(j\colon i+1..n\colon p.j := Ab.j.i / Ab.i.i;\ Ab.j.i := 0)$

Hence, we get the following version of *EliminateRow.i*:

EliminateRow.i, version 4 :–
var *p.j* **is** *real*, *j*: $i+1..n$;
$(j\colon i+1..n\colon p.j := Ab.j.i / Ab.i.i;\ Ab.j.i := 0);$
$(k\colon i+1..n+1\colon (j\colon i+1..n\colon Ab.j.k := Ab.j.k - p.j * Ab.i.k))$

Next we split up the operation *Exchange.i* so that the operations on column *i* are treated separately:

Exchange.i

$\equiv$ [by definition]

$(k\colon i..n+1\colon Ab.i.k, Ab.maxrow.k := Ab.maxrow.k, Ab.i.k)$

$\leqslant$ [unfold sequence one step]

$Ab.i.i, Ab.maxrow.i := Ab.maxrow.i, Ab.i.i;$
$(k\colon i+1..n+1\colon Ab.i.k, Ab.maxrow.k := Ab.maxrow.k, Ab.i.k)$

$\equiv$ [new version]

Exchange.i, version 2

By monotonicity of refinement, we can combine the new versions, to get the following version of the original *Pivot.i* step:

Pivot.i, version 2 :–
var
maxrow **is** *integer*;
p.j **is** *real*, *j*: $i+1..n$;

$$\left.\begin{array}{l} maxrow := i; \\ (j: i+1..n: \textbf{if } |Ab.j.i| > |Ab.maxrow.i| \textbf{ then } maxrow := j \textbf{ fi}); \end{array}\right\} \quad (4)$$

$$\left.\begin{array}{l} Ab.i.i, Ab.maxrow.i := Ab.maxrow.i, Ab.i.i; \\ (k: i+1..n+1: Ab.i.k, Ab.maxrow.k := Ab.maxrow.k, Ab.i.k); \end{array}\right\} \quad (5)$$

$$\left.\begin{array}{l} (j: i+1..n: p.j := Ab.j.i / Ab.i.i;\ Ab.j.i := 0); \\ (k: i+1..n+1: (j: i+1..n: Ab.j.k := Ab.j.k - p.j * Ab.i.k)) \end{array}\right\} \quad (6)$$

here (4) is *FindMaxRow.i*, (5) is *Exchange.i*, *version* 2 and (6) is *EliminateRow.i*, *version* 4. We have also moved the declaration of *p.j* from *EliminateRow.i* to *Pivot.i*. This refinement is justified, because *p.j* does not appear free in *FindMaxRow.i* or *Exchange.i*.

We can rearrange the computation, so that the computation for the first column preceeds the computations for the other columns. This gives us the third version of the pivoting step which is refinement equivalent with the previous version:

Pivot.i, version 3

≡ [merge sequences on k in (5) and (6)]

var
  maxrow **is** *integer*;
  p.j **is** *real*, j: $i+1..n$;

$$\left.\begin{array}{l} maxrow := i; \\ (j: i+1..n: \textbf{if } |Ab.j.i| > |Ab.maxrow.i| \textbf{ then } maxrow := j \textbf{ fi}); \\ Ab.i.i, Ab.maxrow.i := Ab.maxrow.i, Ab.i.i; \\ (j: i+1..n: p.j := Ab.j.i / Ab.i.i;\ Ab.j.i := 0); \end{array}\right\} \quad (7)$$

$$\left.\begin{array}{l} (k: i+1..n+1: Ab.i.k, Ab.maxrow.k := Ab.maxrow.k, Ab.i.k; \\ \qquad\qquad (j: i+1..n: Ab.j.k := Ab.j.k - p.j * Ab.i.k)) \end{array}\right\} \quad (8)$$

≡ [name (7) and (8)]

var
  maxrow **is** *integer*;
  p.j **is** *real*, j: $i+1..n$;
def
  PivotFirst.i :–
    $maxrow := i$;
    (j: $i+1..n$:
      if $|Ab.j.i| > |Ab.maxrow.i|$ **then** $maxrow := j$ **fi**);
    $Ab.i.i, Ab.maxrow.i := Ab.maxrow.i, Ab.i.i$;
    (j: $i+1..n$: $p.j := Ab.j.i / Ab.i.i$; $Ab.j.i := 0$)
  PivotRest.i.k :–
    $Ab.i.k, Ab.maxrow.k := Ab.maxrow.k, Ab.i.k$;
    (j: $i+1..n$: $Ab.j.k := Ab.j.k - p.j * Ab.i.k$)

```
begin
  PivotFirst.i;
  (k: i+1..n+1: PivotRest.i.k)
end
```

≡ [new version]

Pivot.i, version 4

5.3. Refinement step 2: Making pivot operations more independent

We will turn the previous completely sequential algorithm into a form from where it is a short way to a parallel version, by making it possible for several pivoting operations to be active at the same time.

5.3.1. Separate variables per pivoting operation

We have to store the computed pivot elements and their associated maxrow numbers per column so that the operations on the different columns can be done independently of each other. As a first step towards this goal we move the maxrow and pivot declarations out of *Pivot.i*. The *Pivot.i* abstraction is removed as unnecessary. This gives us:

```
Triangularize, version 2 :-
  var
    maxrow is integer;
    p.j is real, j: 2..n;
  def
    PivotFirst.i :-
      maxrow := i;
      (j: i+1..n:
        if |Ab.j.i| > |Ab.maxrow.i| then maxrow := j fi);
      Ab.i.i, Ab.maxrow.i := Ab.maxrow.i, Ab.i.i;
      (j: i+1..n: p.j := Ab.j.i/Ab.i.i; Ab.j.i := 0)
    PivotRest.i.k :-
      Ab.i.k, Ab.maxrow.k := Ab.maxrow.k, Ab.i.k;
      (j: i+1..n: Ab.j.k := Ab.j.k - p.j * Ab.i.k)
  begin
    (i: 1..n-1: PivotFirst.i;
                (k: i+1..n+1: PivotRest.i.k))
  end
```

Observe that the dimension of the array *p.j* was previously dependent on each row index *i* and could therefore vary from 1 to $n-1$. The array is static in the new declaration, and sufficiently large to hold all the elements *p.j*.

We next replace *p.j* **is** *real*, *j*: 2..*n* with *p.i.j* **is** *real*, *i*, *j*: 1..*n* − 1, 2..*n* and *maxrow* **is** *integer* with *maxrow.i* **is** *integer*, *i*: 1..*n* − 1. This replacement is justified in the same way as the replacement of *p* by *p.j* before. This refinement gives us a new version of triangularization, based on new versions of *PivotFirst.i* and *PivotRest.i.k*:

Triangularize, version 3 :–
var
maxrow.i **is** *integer*, *i*: 1..*n* − 1;
p.i.j **is** *real*, *i*,*j*: 1..*n* − 1, 2..*n*;
def
PivotFirst.i :–
maxrow.i := *i*;
(*j*: *i* + 1..*n*:
if |*Ab.j.i*| > |*Ab.*(*maxrow.i*).*i*| **then** *maxrow.i* := *j* **fi**);
Ab.i.i, *Ab.*(*maxrow.i*).*i* := *Ab.*(*maxrow.i*).*i*, *Ab.i.i*;
(*j*: *i* + 1..*n*: *p.i.j* := *Ab.j.i*/*Ab.i.i*; *Ab.j.i* := 0)
PivotRest.i.k :–
Ab.i.k, *Ab.*(*maxrow.i*).*k* := *Ab.*(*maxrow.i*).*k*, *Ab.i.k*;
(*j*: *i* + 1..*n*: *Ab.j.k* := *Ab.j.k* − *p.i.j* * *Ab.i.k*)
begin
(*i*: 1..*n* − 1: *PivotFirst.i*;
(*k*: *i* + 1..*n* + 1: *PivotRest.i.k*)) (9)
end

5.3.2. Introduce iteration counters

As we want the different pivoting operations to proceed in parallel, each column must remember the number of pivoting operations performed. This information is stored in the variable *iter.k* **is** *integer*, *k*: 0..*n* + 2. We also add context assertions on the values of these variables for later use.

The new variables are assigned values as follows. After a complete pivoting step *i*, $i \geq 0$, *iter.i* = *i* + 1 and *iter.j* = *i*, *j* > *i*. Step 0 denotes the initialization. The variables *iter.*0, *iter.*(*n* + 2) are needed in the later refinement step to initiate the first pivoting step respectively mark the end of a pivoting step.

(9)
≡
(*i*: 1..*n* − 1: *PivotFirst.i*;
(*k*: *i* + 1..*n* + 1: *PivotRest.i.k*))

≤ [add fresh variables and context assertions]

var *iter.k* **is** *integer*, *k*: 0..*n* + 2;
*iter.*0 := 1;
(*k*: 1..*n* + 2: *iter.k* := 0);

$$\left.\begin{array}{l}(i\colon 1..n-1\colon iter.i := iter.i+1;\ \{iter.i = i\};\\ \qquad PivotFirst.i;\ iter.i := iter.i+1;\\ \qquad (k\colon i+1..n+1\colon iter.k := iter.k+1;\\ \qquad\qquad \{iter.k = i,\ k > i\};\ PivotRest.i.k);\\ \qquad iter.(n+2) := iter.(n+2)+1;\ \{iter.(n+2) = i\})\end{array}\right\} \quad (10)$$

The correctness of this refinement follows from the fact that the auxiliary variables *iter.k* cannot affect the values of the old variables. It is easily verified that the context assertions act as *skip* statements in this refined version. Observe that we have actually combined two refinement steps here: variable introduction and context introduction.

5.3.3. Make operations depend only on one column index

Our next step is to merge the operations *PivotFirst* and *PivotRest*. The purpose of these refinements is to lead us to an action system with as few different types of actions as possible.

We refine *PivotRest.i.k* with the previously added context assertions:

$\{iter.k = i,\ k > i\};\ PivotRest.i.k$

$\equiv$ [by definition]

$\{iter.k = i,\ k > i\};$
$Ab.i.k,\ Ab.(maxrow.i).k := Ab.(maxrow.i).k,\ Ab.i.k;$
$(j\colon i+1..n\colon Ab.j.k := Ab.j.k - p.i.j * Ab.i.k)$

$\leqslant$ [context introduction]

$\{iter.k = i,\ k > i\};$
$Ab.i.k,\ Ab.(maxrow.i).k := Ab.(maxrow.i).k,\ Ab.i.k;$
$\{iter.k = i,\ k > i\};$
$(j\colon i+1..n\colon Ab.j.k := Ab.j.k - p.i.j * Ab.i.k)$

$\leqslant$ [refinement in context, monotonicity of refinement]

$\{iter.k = i,\ k > i\};$
$Ab.\ (iter.k).k,\ Ab.(maxrow.(iter.k)).k :=$
$\quad Ab.(maxrow.(iter.k)).k,\ Ab.(iter.k).k;$
$\{iter.k = i,\ k > i\};$
$(j\colon iter.k+1..n\colon Ab.j.k := Ab.j.k - p.(iter.k).j * Ab.(iter.k).k)$

$\equiv$ [by definition]

$\{iter.k = i,\ k > i\};\ PivotRest.k$

where $PivotRest.k :- PivotRest.(iter.k).k$.

5.3.4. Merge pivoting operations

Next we note that:

$\{iter.i = i\};\ PivotFirst.i \leqslant PivotColumn.i$

and

$$\{iter.k = i,\ k > i\};\ PivotRest.k \leqslant PivotColumn.k$$

where

$$\begin{array}{l} PivotColumn.k :- \\ \quad \textbf{if } iter.k = k \rightarrow \\ \qquad maxrow.k := k; \\ \qquad (j: k+1..n: \\ \qquad\quad \textbf{if } |Ab.j.k| > Ab.(maxrow.k).k| \textbf{ then } maxrow.k := j \textbf{ fi}); \\ \qquad Ab.k.k,\ Ab.(maxrow.k).k := Ab.(maxrow.k).k,\ Ab.k.k; \\ \qquad (j: k+1..n: p.k.j := Ab.j.k / Ab.k.k;\ Ab.j.k := 0) \\ \quad \square\ iter.k < k \rightarrow \\ \qquad Ab.(iter.k).k,\ Ab.(maxrow.(iter.k)).k := \\ \qquad\quad Ab.(maxrow.(iter.k)).k,\ Ab.(iter.k).k; \\ \qquad (j: iter.k+1..n: \\ \qquad\quad Ab.j.k := Ab.j.k - p.(iter.k).j * Ab.(iter.k).k) \\ \quad \textbf{fi} \end{array}$$

This step can be proved by simplifying *PivotColumn.k* based on the rule

$$\{g_i\};\ \textbf{if } (\square\ k: 1..n: g_k \rightarrow S_k)\ \textbf{fi} \equiv \{g_i\};\ S_i,$$

if $g_i \Rightarrow (\forall j: j \neq i: \neg g_j)$ (observe that $\leqslant$ always holds).

The operations *PivotFirst* and *PivotRest* can now be replaced by *PivotColumn*:

$$\begin{array}{l} \quad (10) \\ \equiv \\ \quad (i: 1..n-1: iter.i := iter.i+1;\ \{iter.i = i\}; \\ \qquad\qquad PivotFirst.i;\ iter.i := iter.i+1; \\ \qquad\qquad (k: i+1..n+1: iter.k := iter.k+1; \\ \qquad\qquad\qquad\qquad \{iter.k = 1,\ k > i\},\ PivotRest.k); \\ \qquad\qquad iter.(n+2) := iter.(n+2)+1;\ \{iter.(n+2) = i\}) \\ \leqslant \quad \text{[refinement in context, monotonicity of refinement]} \\ \quad (i: 1..n-1: iter.i := iter.i+1;\ \{iter.i = i\}; \\ \qquad\qquad PivotColumn.i;\ iter.i := iter.i+1; \\ \qquad\qquad (k: i+1..n+1: iter.k := iter.k+1; \\ \qquad\qquad\qquad\qquad \{iter.k = 1,\ k > i\};\ PivotColumn.k); \\ \qquad\qquad iter.(n+2) := iter.(n+2)+1;\ \{iter.(n+2) = i\}) \end{array}$$

5.3.5. Summing up

Our next refinement is to move the updating of the iteration counters inside the body of *PivotColumn.k*. This refinement is also justified by the context assertions above.

Using monotonicy of refinement we gather all the refinements done so far. We have by now derived the following algorithm for the triangularization phase:

```
Triangularize, version 4 :-
  var
    p.i.j is real, i,j: 1..n-1, 2..n;
    maxrow.i is integer, i: 1..n-1;
    iter.k is integer, k: 0..n+2;
  def
    PivotColumn.k :-
      iter.k := iter.k+1;
      if iter.k = k →
           maxrow.k := k;
           (j: k+1..n:
             if |Ab.j.k| > |Ab.(maxrow.k).k| then maxrow.k := j fi);
           Ab.k.k, Ab.(maxrow.k).k :=
             Ab.(maxrow.k).k, Ab.k.k;
           (j: k+1..n: p.k.j := Ab.j.k/Ab.k.k; Ab.j.k := 0);
           iter.k := iter.k+1
      □ iter.k < k →
           Ab.(iter.k).k, Ab.(maxrow.(iter.k)).k :=
             Ab.(maxrow.(iter.k)).k, Ab.(iter.k).k;
           (j: iter.k+1..n: Ab.j.k := Ab.j.k - p.(iter.k).j * Ab.(iter.k).k);
           if k = n+1 then iter.(n+2) := iter.(n+2)+1 fi
      fi
  begin
    iter.0 := 1;
    (k: 1..n+2: iter.k := 0);
    (i: 1..n-1: PivotColumn.i;
                  (k: i+1..n+1: PivotColumn.k))
  end
```

5.4. Refinement step 3: Distributed triangularization

Let us now distribute the pivoting information among the processes. We assume that the process controlling column k in the matrix is only capable of refering to variables of the columns $k-1$, k, and $k+1$. We further assume that an operation can involve at most two neighbouring columns at any time.

5.4.1. Passing on information

We start by introducing context assertions and variables for passing on information through columns. We add the variables *exchange.k* **is** *integer*, k: $1..n+1$, to record the row that must be exchanged with row k in each pivoting step. We also add variables *pivot.k.j* **is** *real*, k, j: $1..n+1$, $2..n$, to stand for the computed pivot elements

in each pivoting step. The triangularization algorithm is transformed to the following:

```
Triangularize, version 5 :–
  var
    p.i.j is real, i, j: 1..n − 1, 2..n;
    maxrow.i is integer, i: 1..n − 1;
    iter.k is integer, k: 0..n + 2;
    exchange.k is integer, k: 1..n + 1;
    pivot.k.j is real, k,j: 1..n + 1, 2..n;
  def
    PivotColumn.k :–
      iter.k := iter.k + 1;
      if iter.k = k →
           maxrow.k := k;
           (j: k + 1..n:
             if |Ab.j.k| > |Ab.(maxrow.k).k| then maxrow.k := j fi);
           Ab.k.k, Ab.(maxrow.k).k :=
             Ab.(maxrow.k).k, Ab.k.k;
           (j: k + 1..n: p.k.j := Ab.j.k / Ab.k.k; Ab.j.k := 0);
           iter.k := iter.k + 1;
           exchange.k := maxrow.k;
           (j: k + 1..n: pivot.k.j := p.k.j)
      □ iter.k < k →
           exchange.k := exchange.(k − 1);
           (j: iter.k + 1..n: pivot.k.j := pivot.(k − 1).j);
           Ab.(iter.k).k, Ab.(maxrow.(iter.k)).k :=
             Ab.(maxrow.(iter.k)).k, Ab.(iter.k).k;
           (j: iter.k + 1..n: Ab.j.k := Ab.j.k − p.(iter.k).j * Ab.(iter.k).k);
           if k = n + 1 then iter.(n + 2) := iter.(n + 2) + 1 fi
      fi
  begin
    iter.0 := 1;
    (k: 1..n + 2: iter.k := 0);
    {iter.0 = 1 ∧ (k: 1..n + 2: iter.k = 0)};
    (i: 1..n − 1:
      {iter.i = i − 1};
      PivotColumn.i;
      (k: i + 1..n + 1:
        {iter.k = i − 1 ∧ k > i};
        {exchange.(k − 1) = maxrow.(iter.k) ∧
          (j: iter.k + 1..n: pivot.(k − 1).j = p.(iter.k).j)};
        PivotColumn.k))                                          (11)
  end
```

Again only auxiliary variables plus context assertions on them are added. These do not interfere with the old computation.

Our next task is to refine the operation *PivotColumn.k, version* 3 so that the new values are used in computations. We call the *PivotColumn.k* operation of *Triangularize, version* 5 as *PivotColumn.k, version* 3, because it is the result of the third refinement performed for this operation. We have that

$\{iter.i = i-1\};$
PivotColumn.i, version 3

$\leqslant$

PivotColumn.i, version 4

and

$\{iter.k = i-1,\ k > i\};$
$\{exchange.(k-1) = maxrow.(iter.k) \wedge$
$(j: iter.k+2..n: pivot.(k-1).j = p.(iter.k).j)\};$
PivotColumn.k, version 3

$\leqslant$

PivotColumn.k, version 4

where

```
PivotColumn.k, version 4 :-
  iter.k := iter.k + 1;
  if iter.k = k →
        maxrow.k := k;
        (j: k+1..n:
          if |Ab.j.k| > |Ab.(maxrow.k).k| then maxrow.k := j fi);
        Ab.k.k, Ab.(maxrow.k).k :=
          Ab.(maxrow.k).k, Ab.k.k;
        (j: k+1..n: p.k.j := Ab.j.k / Ab.k.k; Ab.j.k := 0);
        iter.k := iter.k + 1;
        exchange.k := maxrow.k;
        (j: k+1..n: pivot.k.j := p.k.j)
  □ iter.k < k →
        exchange.k := exchange.(k-1);
        (j: iter.k+1..n: pivot.k.j := pivot.(k-1).j);
        Ab.(iter.k).k, Ab.(exchange.k).k :=
          Ab.(exchange.k).k, Ab.(iter.k).k;
        (j: iter.k+1..n: Ab.j.k := Ab.j.k - pivot.k.j * Ab.(iter.k).k);
        if k = n+1 then iter.(n+2) := iter.(n+2)+1 fi
  fi
```

The correctness of this step is implied by the previous context assertions.

5.4.2. Change sequence to loop

We now apply the rule which allows us to replace a sequential composition $(i: 1..n: S_i)$ with a loop **do** $(\Box\ i: 1..n: g_i \to S_i)$ **od**:

(11)

$\leqslant$ [remove context assertions]

$iter.0 := 1;$
$(k: 1..n+2: iter.k := 0);$
$\{iter.0 = 1 \wedge (k: 1..n+2: iter.k = 0)\};$
$(i: 1..n-1: PivotColumn.i;$
$\quad (k: i+1..n+1: PivotColumn.k))$

$\leqslant$ [context introduction]

$iter.0 := 1;$
$(k: 1..n+2: iter.k := 0);$
$\{iter.0 = 1 \wedge (k: 1..n+2: iter.k = 0)\};$
$(i: 1..n-1:$
$\quad \{(h: 0..i-1: iter.h = h+1) \wedge (h: i..n+2: iter.h = i-1)\};$
$\quad PivotColumn.i;$
$\quad \{(h: 0..i: iter.h = h+1) \wedge (h: i+1..n+2: iter.h = i-1)\};$
$\quad (k: i+1..n+1:$
$\quad\quad \{(h: i+1..k-1: iter.h \geqslant i) \wedge$
$\quad\quad\quad (h: k..n+2: iter.h = i-1), k > i\},$
$\quad\quad PivotColumn.k))$ (12)

$\leqslant$ [replace inner sequence with a loop, detailed in proof in Appendix B]

$iter.0 := 1;$
$(k: 1..n+2: iter.k := 0);$
$\{iter.0 = 1 \wedge (k: 1..n+2: iter.k = 0)\};$
$(i: 1..n-1:$
$\quad \{(h: 0..i-1: iter.h = h+1) \wedge (h: i..n+2: iter.h = i-1)\};$
$\quad PivotColumn.i;$
$\quad \{(h: 0..i: iter.h = h+1) \wedge (h: i+1..n+2: iter.h = i-1)\};$
$\quad$ **do**
$\quad (\Box\ k: i+1..n+1:$
$\quad\quad (h: i+1..k-1: iter.h \geqslant i) \wedge (h: k..n+2: iter.h = i-1) \to$
$\quad\quad\quad PivotColumn.k)$
$\quad$ **od**)

$\leqslant$ [replace outer sequence with a loop, monotonicity of refinement]

$iter.0 := 1;$
$(k: 1..n+2: iter.k := 0);$
$\{iter.0 = 1 \wedge (k: 1..n+2: iter.k = 0)\};$

$$
\left.\begin{array}{l}
\textbf{do} \\
(\square\, i\colon 1..n-1\colon \\
\quad (h\colon 0..i-1\colon \mathit{iter.h} = h+1) \wedge (h\colon i..n+2\colon \mathit{iter.h} = i-1) \rightarrow \\
\quad\quad \mathit{PivotColumn.i}; \\
\quad\quad \textbf{do} \\
\quad\quad (\square\, k\colon i+1..n+1\colon \\
\quad\quad\quad (h\colon i+1..k-1\colon \mathit{iter.h} \geqslant i) \wedge \\
\quad\quad\quad\quad (h\colon k..n+2\colon \mathit{iter.h} = i-1) \rightarrow \\
\quad\quad\quad\quad\quad \mathit{PivotColumn.k}) \\
\quad\quad \textbf{od}) \\
\textbf{od}
\end{array}\right\} \quad (13)
$$

The proof of correctness of the refinement for the inner sequence in given in Appendix B (Proof A). The other proof (outer sequence) is carried out in a similar manner.

When inspecting the guards of the generated actions we notice that we are able to simplify them with the help of two loop invariants I for the outer loop and J for the inner loop (see Section 3.2 on context dependent replacements):

$$
\begin{array}{l}
I\colon \exists i\colon 1..n\colon (h\colon 0..i-1\colon \mathit{iter.h} = h+1) \wedge \\
\qquad\qquad\qquad (h\colon i..n+2\colon \mathit{iter.h} = i-1) \\
J\colon \exists i, k\colon 1..n, i..n\colon (h\colon i+1..k-1\colon \mathit{iter.h} \geqslant i) \wedge \\
\qquad\qquad\qquad\qquad (h\colon k..n+2\colon \mathit{iter.h} = i-1).
\end{array}
$$

We thus have

$$
\begin{array}{ll}
 & (13) \\
\equiv & \\
 & \textbf{do} \\
 & (\square\, i\colon 1..n-1\colon \\
 & \quad (h\colon 0..i-1\colon \mathit{iter.h} = h+1) \wedge (h\colon i..n+2\colon \mathit{iter.h} = i-1) \rightarrow \\
 & \quad\quad \mathit{PivotColumn.i}; \\
 & \quad\quad \textbf{do} \\
 & \quad\quad (\square\, k\colon i+1..n+1\colon \\
 & \quad\quad\quad (h\colon i+1..k-1\colon \mathit{iter.h} \geqslant i) \wedge (h\colon k..n+2\colon \mathit{iter.h} = i-1) \rightarrow \\
 & \quad\quad\quad\quad \mathit{PivotColumn.k}) \\
 & \quad\quad \textbf{od}) \\
 & \textbf{od} \\
\equiv & \quad [\text{introduce loop invariants}] \\
 & \{I\}; \\
 & \textbf{do} \\
 & (\square\, i\colon 1..n-1\colon \\
 & \quad (h\colon 0..i-1\colon \mathit{iter.h} = h+1) \wedge (h\colon i..n+2\colon \mathit{iter.h} = i-1) \rightarrow \\
 & \quad\quad \mathit{PivotColumn.i};
\end{array}
$$

```
        {J};
        do
        (□ k: i+1..n+1:
            (h: i+1..k−1: iter.h ≥ i) ∧
              (h: k..n+2: iter.h = i−1) →
                PivotColumn.k)
        od)
  od

≡  [change the guards]

  {I};
  do
  (□ i: 1..n−1:
      iter.(i−1) = i ∧ iter.k = i−1 ∧ I →
        PivotColumn.i;
        {J};
        do
        (□ k: i+1..n+1:
            iter.(k−1) ≥ i ∧ iter.k = i−1 ∧ J →
              PivotColumn.k)
        od)
  od                                                        (14)
```

At each iteration there is thus a unique pair of column indices, $l, l+1$, $l = 0, \ldots, n+1$, such that $iter.l > iter.(l+1)$.

5.4.3. Refining atomicity

The actions (14) are in the format which is ready for atomicity refinement. However, the application conditions for this refinement are not satisfied. More precisely, we cannot show that the left movers or right movers commute as required.

Assume that we have refined the atomicity of the previous actions. Consider the pivoting of row i. First perform

$$\{iter.(i-1) = i \wedge iter.i = i-1\};\ PivotColumn.i.$$

Then the operation

$$\{iter.(k-1) \geq i \wedge iter.k = i-1,\ k = i+1\};\ PivotColumn.k$$

can be executed. Thereafter, the two actions involving operations

$$\{iter.i = i+1 \wedge iter.i+1 = i\};\ PivotColumn.i$$

and

$$\{iter.(k-1) \geq i \wedge iter.k = i-1,\ k = i+2\};\ PivotColumn.k$$

are enabled at the same time. These two actions do not, however, commute. If the first mentioned action, start of pivoting of row $i+1$, is activated first, the pivoting

information generated for the pivoting of row i gets destroyed. The updating action must thus be performed before the new pivoting action. Even two subsequent matrix updating actions

$$\{iter.(k-1) \geqslant i \wedge iter.k = i-1,\ k = j, j \in \{i+1, \ldots, n-1\}\};$$
$$PivotColumn.k$$

and

$$\{iter.(k-1) \geqslant i \wedge iter.k = i-1,\ k = j+2, j \in \{i+1, \ldots n-1\}\};$$
$$PivotColumn.k$$

can destroy this information. In this case the (older) update for $k = j+2$ must be performed before the (newer) update for $k = j$. We must therefore carry out one more refinement to make the applicability conditions hold.

Leaning on the loop invariants I and J we can change the guards of the actions into the equivalent guards as follows:

(14)

$\equiv$

$\{I\}$;
do
($\square$ i: 1..$n-1$:
  $iter.(i-1) = i \wedge iter.i = i-1 \wedge I \rightarrow$
    $PivotColumn.i$;
    $\{J\}$;
    do
    ($\square$ k: $i+1..n+1$:
      $iter.(k-1) \geqslant i \wedge iter.k = i-1 \wedge J \rightarrow$
        $PivotColumn.k$)
    od)
od

$\equiv$ [change guards to equivalent ones, remove loop invariants]

do
($\square$ i: 1..$n-1$:
  $iter.(i-1) = i \wedge iter.i = i-1 \wedge iter.(i+1) = i-1 \rightarrow$
    $PivotColumn.i$;
    do
    ($\square$ k: $i+1..n+1$:
      $iter.(k-1) \geqslant i \wedge iter.k = i-1 \wedge iter.(k+1) = i-1 \rightarrow$
        $PivotColumn.k$)
    od)
od

(15)

We are now ready to refine atomicity. We have

(15)

$\leqslant$ [refinement of atomicity, detailed proof in Appendix C]

do
($\Box$ i: $1..n-1$:
  $iter.(i-1) = i \wedge iter.i = i-1 \wedge iter.(i+1) = i-1 \rightarrow$
    PivotColumn.i
($\Box$ i,k: $1..n-1$, $i+1..n+1$:
  $iter.(k-1) \geqslant i \wedge iter.k = i-1 \wedge iter.(k+1) = i-1 \rightarrow$
    PivotColumn.k)
od

The proof of correctness of this step is given in Appendix C (Proof B).

At this point we have managed to produce a distributed algorithm where each operation involves three adjacent columns. We have the following action system:

Triangularize, version 6 :–
var
  $p.i.j$ **is** *real*, i, j: $1..n-1, 2..n$;
  $maxrow.i$ **is** *integer*, i: $1..n-1$;
  $iter.k$ **is** *integer*, k: $0..n+2$;
  $exchange.k$ **is** *integer*, k: $1..n+1$;
  $pivot.k.j$ **is** *real*, k,j: $1..n+1$, $2..n$
def
  PivotColumn.k :–
    $iter.k := iter.k + 1$;
    if $iter.k = k \rightarrow$
        $maxrow.k := k$;
        (j: $k+1..n$:
          if $|Ab.j.k| > |Ab.(maxrow.k).k|$ **then** $maxrow.k := j$ **fi**);
        $Ab.k.k, Ab.(maxrow.k).k := Ab.(maxrow.k).k, Ab.k.k$;
        (j: $k+1..n$: $p.k.j := Ab.j.k / Ab.k.k$; $Ab.j.k := 0$);
        $iter.k := iter.k + 1$;
        $exchange.k := maxrow.k$;
        (j: $k+1..n$: $pivot.k.j := p.k.j$);
    $\Box$ $iter.k < k \rightarrow$
        $exchange.k := exchange.(k-1)$;
        (j: $iter.k+1..n$: $pivot.k.j := pivot.(k-1).j$);
        $Ab.(iter.k).k, Ab.(exchange.k).k :=$
          $Ab.(exchange.k).k, Ab.(iter.k).k$;
        (j: $iter.k+1..n$:
          $Ab.j.k := Ab.j.k - pivot.k.j * Ab.(iter.k).k$);
        if $k = n+1$ **then** $iter.(n+2) := iter.(n+2)+1$ **fi**
    fi

begin
iter.0 := 1;
(*k*: 1..*n* + 2: *iter.k* := 0);
do
(□ *i*: 1..*n* − 1:
iter.(*i* − 1) = *i* ∧ *iter.i* = *i* − 1 ∧ *iter*.(*i* + 1) = *i* − 1 →
PivotColumn.i)
(□ *i,k*: 1..*n* − 1, *i* + 1..*n* + 1:
iter.(*k* − 1) ≥ *i* ∧ *iter.k* = *i* − 1 ∧ *iter*.(*k* + 1) = *i* − 1 →
PivotColumn.k)
od
end (16)

5.5. *Refinement step 4: Merging actions*

Although the previous action system is distributed, the activation of each action requires the co-operation of three column processes. We remove this requirement by introducing the variable *equal.k* **is** *boolean*, *k*: 0..*n* + 1. We maintain the following invariant:

(*k*: 0..*n* + 1: *equal.k* = (*iter.k* = *iter*.(*k* + 1))).

We then have that

(16)
≡
begin
iter.0 := 1;
(*k*: 1..*n* + 2: *iter.k* := 0);
do
(□ *i*: 1..*n* − 1:
iter.(*i* − 1) = *i* ∧ *iter.i* = *i* − 1 ∧ *iter*.(*i* + 1) = *i* − 1 →
PivotColumn.k)
(□ *i*, *k*: 1..*n* − 1, *i* + 1..*n* + 1:
iter.(*k* − 1) ≥ *i* ∧ *iter.k* = *i* − 1 ∧ *iter*.(*k* + 1) = *i* − 1 →
PivotColumn.k)
od
end

≡ [introduce *equal.k* and related context assertions]

begin
iter.0 := 1; *equal*.0 := *false*;
(*k*: 1..*n* + 2: *iter.k* := 0);
(*k*: 1..*n* + 1: *equal.k* := *true*);
{(*k*: 0..*n* + 1: *equal.k* = (*iter.k* = *iter*(*k* + 1)))};

do
($\Box$ i: $1..n-1$:
$iter.(i-1) = i \wedge iter.i = i-1 \wedge iter.(i+1) = i-1 \rightarrow$
$PivotColumn.i$;
$equal.(i-1), equal.i := false, false$)
($\Box$ i, k: $1..n-1, i+1..n+1$:
$iter.(k-1) \geq i \wedge iter.k = i-1 \wedge iter.(k+1) = i-1 \rightarrow$
$PivotColumn.k$;
$equal.(k-1), equal.k :=$
$(iter.(k-1) = iter.k), (k = n+1)$)
od
end

$\equiv$ [change guards to equivalent ones relying on context assertions, remove context assertions]

begin
$iter.0 := 1$; $equal.0 := false$;
(k: $1..n+2$: $iter.k := 0$);
(k: $1..n+1$: $equal.k := true$);
do
($\Box$ i: $1..n-1$:
$iter.i = i-1 \wedge \neg equal.(i-1) \wedge equal.i \rightarrow$
$PivotColumn.i$;
$equal.(i-1), equal.i := false, false$)
($\Box$ i, k: $1..n-1, i+1..n+1$:
$iter.k = i-1 \wedge \neg equal.(k-1) \wedge equal.k \rightarrow$
$PivotColumn.k$;
$equal.(k-1), equal.k :=$
$(iter.(k-1) = iter.k), (k = n+1)$)
od
end (17)

We now move the updating of $equal.k$ into $PivotColumn.k$:

$\{iter.i = i-1\}$; $PivotColumn.i$;
$equal.(i-1), equal.i := false, false$
$\leqslant$
$PivotColumn.i$, *version* 5

and

$\{iter.k = i-1, k > i\}$; $PivotColumn.k$;
$equal.(k-1), equal.k := (iter.(k-1) = iter.k), (k = n+1)$
$\leqslant$
$PivotColumn.k$, *version* 5

where

PivotColumn.k, version 5 :–
  $iter.k := iter.k + 1;$
  if $iter.k = k \rightarrow$
      $maxrow.k := k;$
      $(j: k+1..n:$
        if $|Ab.j.k| > |Ab.(maxrow.k).k|$ **then** $maxrow.k := j$ **fi**$);$
      $Ab.k.k, Ab.(maxrow.k).k := Ab.(maxrow.k).k, Ab.k.k;$
      $(j: k+1..n: p.k.j := Ab.j.k / Ab.k.k;\ Ab.j.k := 0);$
      $iter.k := iter.k + 1;$
      $exchange.k := maxrow.k;$
      $(j: k+1..n: pivot.k.j := p.k.j);$
      $equal.(i-1), equal.i := false, false$
  □ $iter.k < k \rightarrow$
      $exchange.k := exchange.(k-1);$
      $(j: iter.k+1..n: pivot.k.j := pivot.(k-1).j);$
      $Ab.(iter.k).k, Ab.(exchange.k).k :=$
        $Ab.(exchange.k).k, Ab.(iter.k).k;$
      $(j: iter.k+1..n: Ab.j.k := Ab.j.k - pivot.k.j * Ab.(iter.k).k);$
      if $k = n+1$ **then** $iter.(n+2) := iter.(n+2)+1$ **fi**;
      $equal.(k-1), equal.k := (iter.(k-1) = iter.k), (k = n+1)$
  fi

Each action activation now involves variables in two processes only as required.

By carrying out some minor refinements we are able to write the above action system in a simpler form:

(17)
≤ [apply the refinement of *PivotColumn.k* to *PivotColumn.k, version* 5]
do
($\Box$ i: $1..n-1$:
    $iter.i = i-1 \wedge \neg equal.(i-1) \wedge equal.i \rightarrow$
      $PivotColumn.i$)
($\Box$ i, k: $1..n-1, i+1..n+1$:
    $iter.k = i-1 \wedge \neg equal.(k-1) \wedge equal.k \rightarrow$
      $PivotColumn.k$) (18)
od
≡ [regrouping of actions in (18)]
do
($\Box$ k: $1..n-1$:
    $iter.k = k-1 \wedge \neg equal.(k-1) \wedge equal.k \rightarrow$
      $PivotColumn.k$)
($\Box$ k, i: $2..n-1, 1..k-1$:
    $iter.k = i-1 \wedge \neg equal.(k-1) \wedge equal.k \rightarrow$
      $PivotColumn.k$) (19)

$(\Box\ k, i: n..n+1, 1..n-1:$
$\quad iter.k = i-1 \wedge \neg equal.(k-1) \wedge equal.k \rightarrow$
$\quad\quad PivotColumn.k)$
od

$\leqslant$ [merge actions (19)]

do
$(\Box\ k, i: 1..n-1, 1..k:$
$\quad iter.k = i-1 \wedge \neg equal.(k-1) \wedge equal.k \rightarrow$
$\quad\quad PivotColumn.k)$
$(\Box\ k, i: n..n+1, 1..n-1:$
$\quad iter.k = i-1 \wedge \neg equal.(k-1) \wedge equal.k \rightarrow$
$\quad\quad PivotColumn.k)$
od

$\leqslant$ [merge actions by taking the disjunction of guards on i]

do
$(\Box\ k: 1..n-1:$
$\quad (\bigvee i: 1..k: iter.k = i-1) \wedge \neg equal.(k-1) \wedge equal.k \rightarrow$
$\quad\quad PivotColumn.k)$
$(\Box\ k: n..n+1:$
$\quad (\bigvee i: 1..n-1: iter.k = i-1) \wedge \neg equal.(k-1) \wedge equal.k \rightarrow$
$\quad\quad PivotColumn.k)$
od

$\leqslant$ $[(\bigvee i: 1..k: iter.k = i-1) \Leftrightarrow 0 \leqslant iter.k < k$ and
$\quad (\bigvee i: 1..n-1: iter.k = i-1) \Leftrightarrow 0 \leqslant iter.k < n-1]$
do
$(\Box\ k: 1..n-1:$
$\quad 0 \leqslant iter.k < k \wedge \neg equal.(k-1) \wedge equal.k \rightarrow$
$\quad\quad PivotColumn.k)$
$(\Box\ k: n..n+1:$
$\quad 0 \leqslant iter.k < n-1 \wedge \neg equal.(k-1) \wedge equal.k \rightarrow$
$\quad\quad PivotColumn.k)$
od

$\leqslant$ $[min(k, n-1) = k, k = 1, \ldots, n-1$ and $min(k, n-1) = n-1, k = n, n+1]$

do
$(\Box\ k: 1..n+1:$
$\quad 0 \leqslant iter.k < min(k, n-1) \wedge \neg equal.(k-1) \wedge equal.k \rightarrow$
$\quad\quad PivotColumn.k)$
od

5.5.1. Distributed action system

This concludes our derivation of a distributed action system for the triangularization phase. The final parallel algorithm is given below. It is a refinement of the original triangularization phase of the Gaussian elimination algorithm, by transitivity of refinement. We have made some further refinements in the algorithm: the now redundant variables *maxrow.i* and *p.i.j* and assignments to them have been removed and the dimensions of *iter.k* are shrunk. Instead of *maxrow.i* variables a local variable *maxrow* is reintroduced.

```
Triangularize, version 7 :–
  var
    iter.k is integer, k: 1..n+1;
    exchange.k is integer, k: 1..n+1;
    pivot.k.j is real, k, j: 1..n+1, 2..n;
    equal.k is boolean, k: 0..n+1;
  def
    PivotColumn.k :–
      var
        maxrow is integer;
      iter.k := iter.k+1;
      if iter.k = k →
          maxrow := k;
          (j: k+1..n:
            if |Ab.j.k| > |Ab.maxrow.k| then maxrow := j fi);
          Ab.k.k, Ab.maxrow.k :=
            Ab.maxrow.k, Ab.k.k;
          exchange.k := maxrow;
          (j: k+1..n: pivot.k.j := Ab.j.k / Ab.k.k; Ab.j.k := 0);
          iter.k := iter.k+1;
          equal.(k−1), equal.k := false, false
      □ iter.k < k →
          exchange.k := exchange.(k−1);
          (j: iter.k+1..n: pivot.k.j := pivot.(k−1).j);
          Ab.(iter.k).k, Ab.(exchange.k).k :=
            Ab.(exchange.k).k, Ab.(iter.k).k;
          (j: iter.k+1..n:
            Ab.j.k := Ab.j.k − pivot.k.j * Ab.(iter.k).k);
          equal.(k−1), equal.k :=
            (iter.(k−1) = iter.k), (k = n+1)
      fi
  begin
    equal.0 := false;
    (k: 1..n+1: iter.k, equal.k := 0, true);
```

```
do
(□ k: 1..n+1:
    0 ≤ iter.k < min(k,n−1) ∧ ¬equal.(k−1) ∧ equal.k →
      PivotColumn.k)
od
end
```

In [11] this algorithm is further transformed by separating the computation from the variable transmission. In this way a more efficient action system is constructed.

5.5.2. Behaviour of the algorithm

Let us consider a shared action where variables in processes i and j, $i \neq j$, are referenced. We say that the two processes *participate* in the shared action. Assume that there is one process per column. Then the process associated with column k can participate in two different types of actions:

(1) As long as $iter.k < k-1$ the process participates in matrix updating actions concerning pivoting steps $1, \ldots, k-1$. These actions are shared with the column process $k-1$. The process also participates in similar shared actions updating column $k+1$ by turning over the pivoting information to process $k+1$.

(2) As soon as $iter.k = k-1$ process k participates in a shared action with process $k-1$ where the pivoting information concerning column k is computed. Thereafter it participates in exactly one more action: it turns over the pivoting information to column process $k+1$ in a shared action with this process.

6. Concluding remarks

We have shown how the refinement calculus and the action system formalism can be used to derive parallel and distributed algorithms by stepwise refinement starting from a completely sequential algorithm. Each step in the derivation can be formally proved correct, although we did not do this in the derivation above, to keep the presentation within reasonable size.

We notice that we were able to carry out most of the refinement steps by means of the sequential refinement calculus only. This is due to the action system formalism which allows us to design the logical behaviour of the system in terms of sequential nondeterministic statements only. The details of the target system were introduced late in the development, typically in the form of auxiliary variables to guarantee proper sequencing of action activations or to move information around. Atomicity refinement to introduce parallelism was only needed as one of the last steps.

Many of the steps were almost identical. This suggests that we should have a collection of preproved refinement rules which could be applied during the design procedure. Some rules were derived here, other useful rules can be found in [2, 4, 11, 24].

Based on the derivation done here and other nontrivial derivations that we have done, the need for a mechanical tool to support the derivation procedure is apparent. This is a lesson learnt also in other research projects on program transformation systems [26]. A properly designed tool would offer valuable assistance in the often tedious proofs. As a first step towards such a tool we need an environment where it would be easy to administrate a long and complicated derivation. A more sophisticated tool would know the refinement rules "by their names", control the syntax of the programming language used, tell the program constructor what to verify at each step, etc. A really advanced environment might include heuristics or an expert system which gives advice to the programmer on which rules to apply at each step and how to proceed. The design of workstation based tools to support refinements is underway.

The main emphasis in this paper has been to develop a methodology that allows the programmer to start with a traditional, purely sequential algorithm and gradually turn it into a parallel version. We have emphasized the use of a small collection of standard transformation rules in the derivation, in order to parallelize and distribute the algorithm. The derivation thus becomes long, and could possibly be shortened, as also proposed by one referee.

Appendix A. Parallel shared memory algorithm

The sequential algorithm of step 2 can be turned into a parallel action system with shared memory using a similar strategy to the one applied above where the distributed solution with local memory is derived.

We give below only the resulting action system of the parallel algorithm for the triangularization phase.

```
TriangularizeWithSharedMemory :-
  var
    p.k.j is real, k,j: 1..n-1, 2..n;
    maxrow.k is integer, k: 1..n-1;
    iter.k is integer, k: 0..n+1;
  def
    PivotColumn.k :-
      iter.k := iter.k+1;
      if iter.k = k →
          maxrow.k := k;
          (j: k+1..n:
            if |Ab.j.k| > |Ab.(maxrow.k).k| then maxrow.k := j fi);
          Ab.k.k, Ab.(maxrow.k).k :=
            Ab.(maxrow.k).k, Ab.k.k;
          (j: k+1..n: p.k.j := Ab.j.k/Ab.k.k; Ab.j.k := 0);
          iter.k := iter.k+1
```

```
    □ iter.k < k →
        Ab.(iter.k).k, Ab.(maxrow.(iter.k)).k :=
          Ab.(maxrow.(iter.k)).k, Ab.(iter.k).k;
        (j: iter.k + 1..n:
          Ab.j.k := Ab.j.k − p.(iter.k).j * Ab.(iter.k).k)
  fi
begin
  iter.0 := 1;
  (k: 1..n + 1: iter.k := 0);
  do
  (□ k: 1..n + 1:
      0 ≤ iter.k < min(k,n − 1) ∧ iter.(k − 1) > iter.k →
        PivotColumn.k
  od
end
```

The need for shared memory in this solution is apparent since the pivoting information, $p.k.j$, $j = k+1, \ldots, n$ and $maxrow.k$, computed in each column k, $k = 1, \ldots, n-1$, must be available to update columns $l > k$. In other words, each column process must be able to communicate with all other processes. The proper sequencing of the pivoting operations is guaranteed by the $iter.k$ iteration counters.

Appendix B. Proof A (Sequence-to-loop)

Let us control that the requirements of the sequence-to-loop rule are satisfied when refining the inner sequence (12). We first have that

$$
\begin{aligned}
& (12) \\
\equiv\ & \\
& \{(h\colon i+1..k-1\colon iter.h \geq i) \wedge (h\colon k..n+2\colon iter.h = i-1),\ k > i\}; \\
& PivotColumn.k \\
\equiv\ & \quad [k > i,\ iter.k < k] \\
& \{(h\colon i+1..k-1\colon iter.h \geq i) \wedge (h\colon k..n+2\colon iter.h = i-1),\ k > i\}; \\
& PivotColumn.k; \\
& \{(h\colon i+1..k\colon iter.h \geq i) \wedge (h\colon k+1..n+2\colon iter.h = i-1),\ k+1 > i\} \\
\leq\ & \quad [\text{remove context assertions}] \\
& PivotColumn.k; \\
& \{(h\colon i+1..k\colon iter.h \geq i) \wedge (h\colon k+1..n+2\colon iter.h = i-1),\ k > i\}
\end{aligned}
$$

Hence, the first requirement is satisfied.

Let us verify that the second requirement is satisfied. Let therefore

$$g_l = (h\colon i+1..l-1\colon iter.h \geq i) \wedge (h\colon l..n+2\colon iter.h = i-1),$$
$$l > i,\ l \in \{i+1, \ldots, n+1\}.$$

Assume that g_l holds. We have to show that $\neg g_j$, $j \neq l$ where

$$g_j = (h\colon i+1..j-1\colon iter.h \geq i) \wedge (h\colon j..n+2\colon iter.h = i-1),$$
$$j > i,\ j \in \{i+1, \ldots, n+1\}.$$

Assume that even g_j holds. We have two cases:

Case 1: $l > j$. Consider the values of $iter.j, \ldots, iter.l$:

$$g_l \Rightarrow (r\colon j..l-1\colon iter.r \geq i)$$

$$g_j \Rightarrow (r\colon j..l\colon iter.r = i-1).$$

This is a contradiction. The second requirement of the sequence-to-loop rule therefore holds in this case.

Case 2: $j > l$. Consider the values of $iter.l, \ldots, iter.j$:

$$g_l \Rightarrow (r\colon l..j\colon iter.r = i-1)$$

$$g_j \Rightarrow (r\colon l..j-1\colon iter.r \geq i).$$

Even this is a contradiction.

Refining the inner sequence to a loop is thus done properly. The correctness of the outer loop is shown accordingly (we skip the details here).

Appendix C. Proof B (Atomicity refinement)

The correctness of the refinement of atomicity is shown by induction on i. The basis step is for $i = n-1$ (observe that the conditions for atomicity refinement permit us to make this refinement only one action at a time). In each step we have to show that (1) an old action cannot enable or disable a new action except the initialization of the refining actions and that no new action is enabled initially, except this initialization. We must then also show that (2) the initialization is never enabled when some other new action is enabled. Finally, we have to show that (3) the old actions that can be enabled when some new action is enabled satisfy the conditions (3.1)-(3.3) in Section 3.4.

We name the actions as follows:

(1) The unrefined actions will be called $A_1, \ldots, A_{n-1}$.
(2) The refined pivot element computation actions will be called $A_{1,1}, \ldots, A_{n-1,n-1}$.
(3) The refined actions updating the coefficient matrix will be called $A_{1,2}, \ldots, A_{1,n+1}, A_{2,3}, \ldots, A_{2,n+1}, \ldots, A_{n-1,n}, A_{n-1,n+1}$.

In the basis step the old actions are the original actions A_i, $i = 1, \ldots, n-2$. The initial action is the pivot element computation $A_{n-1,n-1}$. The other new actions are

$A_{n-1,k}$, $k = n, n+1$. We show that the applicability conditions hold in this step:

(1) No new action is initially enabled (except $A_{n-1,n-1}$ if $n = 2$), because *iter.k* equals 0 for $k = 1, \ldots, n+2$. Each old action updates *iter.*$(n-1)$ in this way affecting the enabling of the new actions. However, an old action can only enable the initialization action.
(2) The initialization action turns itself off once it has been executed. As a new action cannot be enabled unless the initialization has been done and does not itself enable the initialization action, it follows that the initialization is not enabled when any new action is enabled.
(3) No old action can be enabled when one of the new actions is enabled, so the conditions (3.1)-(3.3) hold trivially.

In the induction step we consider refining atomicity of the ith original pivoting action assuming that the atomicity of the steps $i+1, \ldots, n-1$ has already been refined. The old actions are now the original pivoting actions A_k, $k = 1, \ldots, i-1$, and the already refined actions $A_{k,l}$, $k = i+1, \ldots, n-1$, $l = k, \ldots, n+1$. The new actions are the new initial pivot element computation $A_{i,i}$ and the matrix updating operations $A_{i,k}$, $k = i+1, \ldots, n+1$. We have that:

(1) The original old actions cannot enable or disable a new action except the initialization, by a similar argument as above. Also the already refined old actions cannot enable or disable the initialization, because they do not refer to same variables. The refined old actions update *iter.k* for $k = i+1, \ldots, n+2$. They cannot, however, enable or disable a new action, because these new actions have to be activated before the old actions.
(2) The initialization action turns itself off once it has been executed. As a new action cannot be enabled unless the initialization has been done and does not enable the initialization, it follows that the initialization is not enabled when any new action is enabled.
(3) None of the original old actions can be enabled when a new action is enabled. Also the pivot element computation $A_{j,j}, j = i+1, \ldots, n-1$, cannot be enabled at the same time as the matrix updating $A_{i,j}$. Similarly, the matrix updating $A_{k,j-1}, j = i+2, \ldots, n+1$, $k > i$ cannot be enabled when the matrix updating $A_{i,j}$ is enabled. Hence, the only actions that can be enabled are the rest of the already refined old actions. We will consider these actions as right movers. We must thus prove that (3.2) and (3.3) hold. The latter is easily seen to be the case, as each right mover increments the values of *iter.k* which have the upper limits $min(k+1, n-1)$. For the condition (3.2) we have to show that each right mover that can be followed by a new matrix updating operation $A_{i,j}$, $j = i+1, \ldots, n+1$, commutes right with this new action. We have that the right mover operation $A_{k,k}$, $k > i$, can be followed by the new matrix updating operation $A_{i,j}$, $j > k+1$. However, these two actions do not have any variables in common, so they commute. Similarly, a right mover operation

$A_{k,l}$, $k > i$, $l > k$ can be followed by a new operation $A_{i,j}$, $j > l+2$. Again these two actions do not have any variables in common, so they commute.

Acknowledgement

We would like to thank Viking Högnäs, Hong Shen and Joakim von Wright for careful reading of an earlier draft of the paper. The work reported here was supported by the Academy of Finland and The Technology Developement Centre of Finland.

References

[1] R.J.R. Back, On the correctness of refinement in program development, Ph.D. Thesis, Rept. A-1978-4, Department of Computer Science, University of Helsinki, Finland (1978).
[2] R.J.R. Back, Correctness preserving program refinements: Proof theory and applications, Mathematical Centre Tracts 131, Mathematical Centre, Amsterdam, Netherlands (1980).
[3] R.J.R. Back, Procedural abstraction in the refinement calculus, *Åbo Akademi, Reports on Computer Science and Mathematics Ser. A* **55** (1987).
[4] R.J.R. Back, A calculus of refinements for program derivations, *Acta Inform.* **25** (1988) 593–624.
[5] R.J.R. Back, Refining atomicity in parallel algorithms, in: *Proceedings PARLE: Parallel Architectures and Languages Europe II: Parallel Languages*, Lecture Notes in Computer Science (Springer, Berlin, 1989).
[6] R.J.R. Back, E. Hartikainen and R. Kurki-Suonio, Multi-process handshaking on broadcasting networks, *Åbo Akademi, Reports on Computer Science and Mathematics Ser. A* **42** (1985).
[7] R.J.R. Back and R. Kurki-Suonio, Decentralization of process nets with centralized control, in: *Proceedings 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Montreal, Que. (1983) 131–142; also *Distributed Comput.* **3** (1989).
[8] R.J.R. Back and R. Kurki-Suonio, A case study in constructing distributed algorithms: Distributed exchange sort, in: *Proceedings Winter School on Theoretical Computer Science*, Lammi, Finland (1984) 1–33.
[9] R.J.R. Back and R. Kurki-Suonio, Co-operation in distributed systems using symmetric multi-process handshaking, *Åbo Akademi, Reports on Computer Science and Mathematics Ser. A* **34** (1984).
[10] R.J.R. Back and R. Kurki-Suonio, Distributed co-operation with action systems, *ACM Trans. Programming Languages Syst.* **10** (4) (1988) 513–554.
[11] R.J.R. Back and K. Sere, An exercise in deriving parallel algorithms: Gaussian elimination, *Åbo Akademi, Reports on Computer Science and Mathematics Ser. A* **65** (1988).
[12] R.J.R. Back and K. Sere, Deriving an implementation of action systems in occam, Manuscript.
[13] K.M. Chandy and J. Misra, An example of stepwise refinement of distributed programs: Quiescence detection, *ACM Trans. Programming Languages Syst.* **8** (3) (1986) 326–343.
[14] K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation* (Addison-Wesley, Reading, MA, 1988).
[15] E.W. Dijkstra, *A Discipline of Programming* (Prentice-Hall International, Englewood Cliffs, NJ, 1976).
[16] E.W. Dijkstra, L. Lamport, A.J. Martin and C.S. Scholten, On-the-fly garbage collection: An exercise in cooperation, *Comm. ACM* **21** (1978) 966–975.
[17] M. Evengelist, V.Y. Shen, I. Forman and M. Graf, Using Raddle to design distributed systems, MCC Tech. Rept. No. STP-285-87 (1987).
[18] C.A.R. Hoare, Communicating sequential processes, *Comm ACM* **21** (8) (1978) 666–677.
[19] C.A.R. Hoare, *Communicating Sequential Processes* (Prentice-Hall International, Englewood Cliffs, NJ, 1985).

[20] INMOS Ltd., *Occam Programming Manual* (Prentice-Hall International, Englewood Cliffs, NJ, 1984).
[21] A.R. Martin and J.V. Tucker, The concurrent assignment representation of synchronous systems, in: *Proceedings PARLE, Parallel Architectures and Languages Europe II: Parallel Languages*, Lecture Notes in Computer Science **256** (Springer, Berlin, 1987) 369-386.
[22] C. Morgan, Data refinement by miracles, *Inform. Process. Lett.* **26** (1987/88) 243-246.
[23] C. Morgan, The specification statement, *ACM Trans. Programming Languages Syst.* **10** (3) (1988) 403-419.
[24] C. Morgan, K.A. Robinson and P.H.B. Gardiner, On the refinement calculus, Draft (1988).
[25] J.M. Morris, A theoretical basis for stepwise refinement and the programming calculus, *Sci. Comput. Programming* **9** (1987) 287-306.
[26] H. Partsch and R. Steinbrüggen, Program transformation systems, *ACM Comput. Surv.* **15** (3) (1983) 199-236.
[27] A. Pnueli, Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends, in: J.W. de Bakker, W.P. de Roever and G. Rozenberg, eds., *Current Trends in Concurrency*, Lecture Notes in Computer Science **224** (Springer, Berlin, 1986).
[28] S. Ramesh and S.L. Mehndiratta, A methodology for developing distributed programs, *IEEE Trans. Software Engrg.* **13** (8) (1987) 967-976.
[29] W. Reisig, *Petri Nets, an Introduction*, EATCS Monographs on Theoretical Computer Science (Springer, Berlin, 1985).
[30] A.U. Shankar and S.S. Lam, Time-dependent distributed systems: Proving safety, liveness and real-time properties, *Distributed Comput.* **2** (2) (1987) 61-79.