Constructing some Distributed Programs

H. Peter Hofstee
Department of Computer Science
California Institute of Technology

Contents

0.1 Introduction

This report contains a collection of papers on constructing concurrent algorithms. Besides presenting a series of new algorithms, the papers have been written to gain better understanding of methods for proving, and thus for constructing, concurrent algorithms. Methods for proving correctness of sequential algorithms, and techniques that use the construction of such a proof to construct algorithms themselves have been well developed. These proofs can be made entirely formal, an indication that there is indeed no missing knowledge. The proofs for concurrent programs lack this degree of formalism. Throughout this paper a set of correctness criteria is used that is somewhat loosely based on an interleaving semantics, and the properties of channels, but that lacks complete rigor. Existing proof methods that are directly based on the interleaving semantics and that are therefore completely rigorous within this model, are too cumbersome to use for any but the very smallest programs. Other proof theories deal with sequential programs that simulate concurrent behavior, but there is no direct correspondence between these programs and an efficient concurrent implementation. A third goal of this work is to gain understanding of programs for loosely coupled message passing fine grain multicomputers. Unlike single instruction multiple data (SIMD) machines, synchronization is not implied, but has to be programmed. It is unlikely that all processors will perform the same task in synchrony; if they would, a SIMD machine would be more suitable for the task. One would like to develop algorithms that can be used in a wide variety of contexts. The algorithm should therefore not depend on detailed analysis of a particular network, or on information about other tasks performed by the processors. These considerations make the cost of synchronization hard to predict, therefore synchronization is kept to a minimum in the algorithms developed here. It also seems unlikely that deterministic programs, whose execution does not depend on properties of the network, or the workload of the processors, are most efficient. The algorithms presented in chapter two and three are therefore highly nondeterministic.

The method followed in these papers is the following. To construct a concurrent solution, a sequential solution is given first. A decision is made about the distribution of the variables and the sequential solution is transformed to minimize the use of statements that refer to variables that will be allocated to different processes. In the following step processes and communication are introduced. Finally the concurrent solution is transformed to a more efficient one or, in one case, to add termination detection.

In the first paper a concurrent implementation of exchange sort is derived. In the second paper a load balancing algorithm is constructed. In the third paper a transformation to a concurrent algorithm for a class of sequential algorithms is presented. New algorithms for distributed sorting, and a more general solution to the load balancing problem are presented

as examples. The final paper gives results of running the algorithms on the MOSAIC and S2010 multicomputers.

Since this collection of papers is also my masters thesis, I should explain briefly what part of the work is mine. Chapter one has been written by Jan van de Snepscheut. Alain Martin started this topic with the presentation for two concurrent processes. I came up with a solution for N processes, and a correctness argument, before I knew much about proofs of concurrent algorithms. The proof of the algorithm in the paper is largely Jan's work. The more efficient solution and its proof are Jan's work entirely. I wrote the paper in chapter two, but when a proof is elegant you can bet that Jan rewrote mine. The postcondition and the solution strategy was developed in group meetings by Jan, Johan Lukkien and myself. Chapter three is my work, but I have many people to thank as can be seen in the acknowledgements of that paper. It replaces an unpublished paper on nondeterministic distributed sorting algorithms. Chapter four represents a small portion of countless runs of my programs, written in Jan's concurrent Pascal and the message passing C of Chuck Seitz's group. A paper on a somewhat different application of program transformations, a derivation of asynchronous memories, was part of the masters work, but has not been included in this report.

1

Distributed Sorting

- Published previously in *Science of Computer Programming* 15, (1990) 119-133

H. Peter Hofstee, Alain. J. Martin, and Jan L. A. van de Snepscheut

ABSTRACT In this paper we present a distributed sorting algorithm, which is a variation on exchange sort, i.e., neighboring elements that are out of order are exchanged. We derive the algorithm by transforming a sequential algorithm into a distributed one. The transformation is guided by the distribution of the data over processes. First we discuss the case of two processes, and then the general case of one or more processes. Finally we propose a more efficient solution for the general case.

1.1 Program notation

For the sequential part of the algorithms, we use a subset of Edsger W. Dijkstra's guarded command language [1]. For (sequential) statements $S0$ and $S1$, statement $S0||S1$ denotes their concurrent execution. The constituents $S0$ and $S1$ are then called processes. The statements may share variables (cf. [6]). We transform our algorithms in such a way, however, that the final code contains no shared variables and all synchronization and communication is performed by message passing. The semantics of the communication primitives is as described in [5]. The main difference with C.A.R. Hoare's proposal in [3] is in the naming of channels rather than processes. In [4], the same author proposes to name channels instead of processes in communication commands, but differs from our notation by using one name per channel instead of our two: output command $R!E$ in one process is paired with input command $L?v$ in another process by declaring the pair (R, L) to be a channel between the two processes. Each channel is between two processes only. When declaring (R, L) to be a channel, we write the name on which the output actions are performed first and the name on which the input actions are performed last.

For an arbitrary command A, let $\mathbf{c}\,A$ denote the number of completed A actions, i.e., the number of times that command A has been executed since initiation of the program's execution. The *synchronization requirement* (cf.

[5]) fulfilled by a channel (R, L) is that

$$\mathbf{c}\, R = \mathbf{c}\, L$$

holds at any point in the computation.
Note It is sometimes attractive to weaken the synchronization requirement by putting some bound on $\mathbf{c}\, R - \mathbf{c}\, L$. This may be a lower bound only, or both a lower and an upper bound. The maximum difference is then called the *slack*, since it indicates how far the synchronized processes can get out of step. The use of a nonzero slack sometimes leads to minor complications in proofs and definitions, and is not pursued here.
The execution of a command results either in the completion of the action or in its suspension when its completion would violate the synchronization requirement. From suspension until completion an action is *pending* and the process executing the action is delayed. We introduce boolean $\mathbf{q}\, A$ equal to the predicate "an A action is pending". The *progress requirement* states that actions are suspended only if their completion would violate the synchronization requirement, i.e., channel (R, L) satisfies

$$\neg\mathbf{q}\, R \;\vee\; \neg\mathbf{q}\, L \qquad .$$

The nth R action is said to match the nth L action. The completion of a matching pair of actions is called a *communication*. The *communication requirement* states that execution of matching actions $R!E$ and $L?v$ amounts to the assignment $v := E$.

1.2 A small/large sorter for two bags

Given are two finite, nonempty bags of integers. The integers in the two bags are to be rearranged such that one bag is dominated by the other bag, i.e., no element of the first bag exceeds any element of the second bag. The number of elements of each of the two bags may not be changed.

We use the following notation. The two bags to be sorted are $b0$ and $b1$; their initial values are $B0$ and $B1$ respectively. For bag b, $\#b$ denotes the number of elements in b. Bag union and difference are denoted by $+$ and $-$ respectively. The number of times that a number x occurs in the bag union $b0 + b1$ is the number of occurrences of x in $b0$ plus the number of occurrences in $b1$. The number of occurrences of x in $b0 - b1$ is the number of occurrences of x in $b0$ minus the number of occurrences in $b1$, and is well-defined only if the latter difference is nonnegative. We do not distinguish between elements and singleton bags.

Postcondition Z of the distributed sorting program is concisely written as follows.

The first two conjuncts express that the size of the two bags is unaffected, the third conjunct expresses that the elements involved remain the same,

$$Z : \#b0 = \#B0 \ \wedge \ \#b1 = \#B1 \ \wedge \ b0 + b1 = B0 + B1 \ \wedge$$
$$\max(b0) \leq \min(b1)$$

and the fourth conjunct expresses that $b0$ is dominated by $b1$. Notice that $\max(b0) < \min(b1)$ is a stronger requirement: in fact it is so strong that it cannot be established in general.

The problem can simply be solved by repeatedly exchanging the maximum element of $b0$ and the minimum element of $b1$ until postcondition Z is established. This amounts to selecting the first three conjuncts of Z as invariant

$$\#b0 = \#B0 \ \wedge \ \#b1 = \#B1 \ \wedge \ b0 + b1 = B0 + B1$$

and the negation of the last conjunct of Z as guard of the repetition. The program is

do $\max(b0) > \min(b1) \rightarrow$
$$b0, b1 := b0 + \min(b1) - \max(b0), b1 + \max(b0) - \min(b1)$$
od

The invariant is vacuously true upon initialization since then $b0 = B0 \ \wedge \ b1 = B1$. The invariant is maintained by the exchange statement, independently of the guard. Upon termination $\max(b0) \leq \min(b1)$ holds, which in conjunction with the invariant implies postcondition Z. We are left with the easy task of proving termination. Let variant function s be the sum of the elements in $b0$ minus the sum of the elements in $b1$. Since $b0$ and $b1$ are finite bags with fixed union, s is bounded from below. On account of the guard, every exchange decreases the sum of the elements in $b0$ and increases the sum of the elements in $b1$, and thereby decreases s. Hence, the loop terminates.

1.3 Program transformation

We shall now transform the program, under invariance of its semantics, so as to partition it into two sets of (almost) noninterfering statements. We introduce fresh variables both for this purpose and for avoiding repeated

evaluation of $\max(b0)$ and $\min(b1)$. When we have two sets of noninterfering statements they can be executed by two processes, which is what we aim at. The interference that remains translates into communication or synchronization actions. Introducing M and m to avoid reevaluation of max and min, and copies LM and rm to reduce interference yields the program in Figure 1.1.

$M, m := \max(b0), \min(b1);$
$rm, LM := m, M;$
do $guard \rightarrow b0, b1 := b0 + rm - M, b1 + LM - m;$
$M, m := \max(b0), \min(b1);$
$rm, LM := m, M$
od

FIGURE 1.1.

Notice that guard $\max(b0) > \min(b1)$ can be rewritten in many ways, including $M > rm$ and $LM > m$. In Figure 1.1, we have not made a choice yet, and both rewrites will be used later (which is the reason for not writing down a specific guard here). The bag differences in $b0 + rm - M$ and $b1 + LM - m$ are well-defined since M is an element of $b0$ and m is an element of $b1$. Apart from the concurrent assignment $rm, LM := m, M$ we have partitioned the program into two sets of noninterfering statements. Since the order of noninterfering statements can be swapped freely, we can modify the program slightly so as to group together the actions on $b0, M, rm$ and the actions on $b1, m, LM$. We obtain Figure 1.2 in which a suggestive layout has been used.

Now, assume that we can split the action $rm := m \parallel LM := M$ into two concurrent parts, X and Y say, such that $\mathbf{c}\, X = \mathbf{c}\, Y$ and $\neg\mathbf{q}\, X \vee \neg\mathbf{q}\, Y$ hold, and such that the completion of X and Y is equivalent to $rm := m \parallel LM := M$. We may rewrite the program from Figure 1.2 into $(p0 \parallel p1)$ as given in Figure 1.3. Notice that we have used both ways of rewriting the guard mentioned above.

The correctness of the program in Figure 1.3 can be proved in two ways. We may either prove the correctness of the transformation, or we may prove the correctness of the program in Figure 1.3 directly. Proving the correctness of the transformation is the more elegant (and slightly easier) of the two. Yet we give a direct proof of the program's correctness, because it comes closer to suggesting the generalization to any number of processes.

$$
\begin{array}{lcl}
M := \max(b0) & \| & m := \min(b1); \\
rm := m & \| & LM := M; \\
\multicolumn{3}{c}{\textbf{do } guard \rightarrow} \\
(b0 := b0 + rm - M;\ \ M := \max(b0)) & \| & (b1 := b1 + LM - m;\ \ m := \min(b1)); \\
rm := m & \| & LM := M \\
\multicolumn{3}{c}{\textbf{od}}
\end{array}
$$

FIGURE 1.2.

$$
\begin{array}{lcl}
p0 & \equiv & M := \max(b0);\ \ X; \\
 & & \textbf{do } M > rm \rightarrow b0 := b0 + rm - M;\ \ M := \max(b0);\ \ X \textbf{ od} \\
p1 & \equiv & m := \min(b1);\ \ Y; \\
 & & \textbf{do } LM > m \rightarrow b1 := b1 + LM - m;\ \ m := \min(b1);\ \ Y \textbf{ od}
\end{array}
$$

FIGURE 1.3.

We postulate that P is an invariant of the distributed program.

$$P: \quad \max(b0) = M = LM \ \wedge \ \min(b1) = m = rm \ \wedge$$

$$\#b0 = \#B0 \ \wedge \ \#b1 = \#B1 \ \wedge \ b0 + b1 = B0 + B1$$

What do we mean by claiming that P is an invariant of $(p0 \parallel p1)$? Both $p0$ and $p1$ contain a loop and by invariant we mean in this case that P holds when both processes have completed the initialization (and no further actions), and that P is maintained if both processes perform one step of the loop. Since initialization and loop body end with action X in $p0$ and action Y in $p1$, and since we have $\mathbf{c}\,X = \mathbf{c}\,Y$, this makes sense. Notice that, for example, we do not claim that P holds if $p0$ has completed X, whereas $p1$ has completed Y and also the subsequent update of $b1$. In order to check the invariance, we have to verify

- $\{true\}\ (M := \max(b0);\ \ X) \parallel (m := \min(b1);\ \ Y)\ \{P\}$
- $P \ \Rightarrow \ (M > rm \ \equiv \ LM > m)$
- $\{P \ \wedge \ M > rm \ \wedge \ LM > m\}$
  $(b0 := b0 + rm - M;\ \ M := \max(b0);\ \ X) \parallel$
  $(b1 := b1 + LM - m;\ \ m := \min(b1);\ \ Y)$
  $\{P\}$.

All three follow from the choice of P and the assumptions on X and Y.

We are left with the task of providing X and Y in terms of the commands that we have at our disposal. Using channels (R, L) and (l, r), with zero slack, we may write

$$X \ \equiv \ R!M;\ \ r?rm$$

$$Y \ \equiv \ L?LM;\ \ l!m$$

Since $\mathbf{c}\,X = \mathbf{c}\,r$ and $\mathbf{c}\,Y = \mathbf{c}\,l$ by construction, and since $\mathbf{c}\,r = \mathbf{c}\,l$ by definition, we have $\mathbf{c}\,X = \mathbf{c}\,Y$. Actions X and Y may be suspended on either channel, hence

$$\mathbf{q}\,X \ \equiv \ (\mathbf{c}\,R = \mathbf{c}\,r \ \wedge \ \mathbf{q}\,R) \ \vee \ (\mathbf{c}\,R = \mathbf{c}\,r + 1 \ \wedge \ \mathbf{q}\,r)$$

$$\mathbf{q}\,Y \ \equiv \ (\mathbf{c}\,L = \mathbf{c}\,l \ \wedge \ \mathbf{q}\,L) \ \vee \ (\mathbf{c}\,L = \mathbf{c}\,l + 1 \ \wedge \ \mathbf{q}\,l)$$

We calculate

$$
\begin{array}{ll}
 & \mathbf{q}\,X \ \wedge \ \mathbf{q}\,Y \\
= & \qquad \{\mathbf{c}\,R = \mathbf{c}\,L, \ \mathbf{c}\,r = \mathbf{c}\,l\} \\
 & (\mathbf{c}\,R = \mathbf{c}\,r = \mathbf{c}\,L = \mathbf{c}\,l \ \wedge \ \mathbf{q}\,R \ \wedge \ \mathbf{q}\,L) \ \vee \\
 & (\mathbf{c}\,R = \mathbf{c}\,r + 1 = \mathbf{c}\,l = \mathbf{c}\,l + 1 \ \wedge \ \mathbf{q}\,r \ \wedge \ \mathbf{q}\,l) \\
\Rightarrow & \\
 & (\mathbf{q}\,R \ \wedge \ \mathbf{q}\,L) \ \vee \ (\mathbf{q}\,r \ \wedge \ \mathbf{q}\,l) \\
= & \qquad \{\neg\mathbf{q}\,R \ \vee \ \neg\mathbf{q}\,L, \neg\mathbf{q}\,r \ \vee \ \neg\mathbf{q}\,l\} \\
 & \mathbf{false}
\end{array}
$$

i.e., we have $\neg\mathbf{q}\,X \ \vee \ \neg\mathbf{q}\,Y$ as required. From the communication requirement it follows that $X \parallel Y$ is equivalent to $rm := m \parallel LM := M$.

The following, more symmetric, version of X and Y also meets the requirements.

$$X \ \equiv \ R!M \parallel r?rm$$

$$Y \ \equiv \ L?LM \parallel l!m$$

The above two versions are also correct if the slack is positive. The version

$$X \ \equiv \ R!M; \ r?rm$$

$$Y \ \equiv \ l!m; \ L?LM$$

is correct only if the slack is positive.

Observe that the verification of the correctness of the program fits the following pattern. We postulate an invariant and show that it holds in the initial state and is not falsified by an iteration of the loop. We provide a variant function that is bounded from below and decreases with each iteration of the loop. Because the variant function is integer valued, this implies that the loop terminates. Upon termination we have the truth of the invariant and the falsity of the loop's guard, and we show that this combination implies the postcondition. So much is standard practice in the case of sequential programs. What we add for our distributed programs is the proof of absence of deadlock. Deadlock occurs if one of two processes connected by a channel initiates a communication along the channel and the other process does not. We, therefore, show that mutual communications are initiated under the same condition.

1.4 More bags

The problem is generalized as follows. Given is a finite sequence of (one or more) finite nonempty bags and a linear array of processes, each of which

contains one of the bags and communicates with neighbors in the array to sort the bags in such a way that each bag is dominated by the next bag in the sequence, and such that the size of the bags remains constant.

The generalized problem is significantly different from the two-bag version in the following sense. Consider sequence ABC of three bags. If A is dominated by B but B is not dominated by C then an exchange of elements between B and C may cause B to no longer dominate A, i.e., it may necessitate an exchange between A and B. This shows that the process that stores A cannot be terminated when A is dominated by B. The proper thing to do is to terminate a process when the bag it stores is dominated by all bags to the right of it and dominates all bags to the left of it. The two algorithms that follow are both based on exchanging elements of neighboring bags, and termination detection while ensuring progress is the hard part of the problem.

In order to avoid excessive use of subscripts, we use the following notation. For some anonymous process, b is the bag it stores with initial value B, rb is the union of all bags to its right with initial value rB, and lb is the union of all bags to its left with initial value lB. Notice that lB and rB are the empty bag $\emptyset$ for the leftmost and rightmost processes respectively. The required postcondition of the program can be written as a conjunction of terms, one for each process, viz.

$$\max(lb) \leq \min(b) \quad \wedge \quad \max(b) \leq \min(rb) \quad .$$

We find it more attractive to rewrite this into

$$\max(lb) \leq \min(b + rb) \quad \wedge \quad \max(lb + b) \leq \min(rb)$$

since the first term expresses that domination has been achieved between the union of all bags to the left and the remaining bags, and the second term does so for all bags to the right. The invariant is obtained by introducing a variable for each of the four quantities involved, and by retaining the size restriction on the bags. Hence, the invariant of the distributed program is the conjunction of a number of terms, one for each process. Each such term is

$$\begin{aligned} P :& \max(lb) = LM \quad \wedge \quad \max(lb + b) = M \\ & \wedge \quad \min(rb) = rm \quad \wedge \quad \min(b + rb) = m \\ & \wedge \quad \#b = \#B \quad \wedge \quad lb + b + rb = lB + B + rB \end{aligned}$$

where $\max(\emptyset) = -\infty$ and $\min(\emptyset) = +\infty$. First we concentrate on the statements that initialize the variables such that P holds. Maxima are simply propagated from left to right, and minima from right to left.

$$(L?LM;\ \ M := \max(b + LM);\ \ R!M)$$
$$\|\ (r?rm;\ \ m := \min(b + rm);\ \ l!m)$$

Action $L?LM$ is understood to be $LM := -\infty$ for the leftmost process, and $r?rm$ is understood to be $rm := +\infty$ for the rightmost process. Action $l!m$ is understood to be *skip* for the leftmost process, and $R!M$ is understood to be *skip* for the rightmost process. These conventions can be implemented with dummy processes next to the two extreme processes, or with conditional statements in the two extreme processes. Next we concentrate on the loop, i.e., we concentrate on maintaining the invariant. Observe that an element from lb should be exchanged with an element from $b + rb$ if $\max(lb) > \min(b + rb)$, i.e., if $LM > m$. Like in the case of two bags, the maximum element from lb is exchanged with the minimum element from $b + rb$. Similarly, the minimum element from rb is exchanged with the maximum element from $lb + b$ if $\max(lb + b) > \min(rb)$, i.e., if $M > rm$. This suggests the program shown in Figure 1.4.

$(L?LM;\ \ M := \max(b + LM);\ \ R!M)\ \|\ (r?rm;\ \ m := \min(b + rm);\ \ l!m);$
do $LM > m\ \ \wedge\ \ M > rm \rightarrow$
$\quad b := b + LM - M + rm - m;$
$\quad (L?LM;\ \ M := \max(b + LM);\ \ R!M)\ \|\ (r?rm;\ \ m := \min(b + rm);\ \ l!m)$
$[\!]\ LM > m\ \ \wedge\ \ M \leq rm \rightarrow$
$\quad b := b + LM - m;$
$\quad (L?LM;\ \ M := \max(b + LM))\ \|\ (m := \min(b);\ \ l!m)$
$[\!]\ LM \leq m\ \ \wedge\ \ M > rm \rightarrow$
$\quad b := b + rm - M;$
$\quad (M := \max(b);\ \ R!M)\ \|\ (r?rm;\ \ m := \min(b + rm))$
od

FIGURE 1.4.

We prove the correctness of this algorithm. Consider two processes that are neighbors in the linear array. Bag $lb + b$ in the left process is bag b in the right process, hence M in the left process has the same value as LM in the

right process. Similarly, bag rb in the left process is bag $b + rb$ in the right process, hence rm in the left process has the same value as m in the right process. The left process initiates a communication with the right process if and only if $M > rm$ holds, and the right process initiates a communication with the left process if and only if $LM > m$ holds. Consequently, the two processes initiate their mutual communications under the same condition, which excludes deadlock.

Because of the above-established correspondence between M and rm in one process and LM and m in its righthand neighbor, updating the bags leaves the union of all bags constant provided that an element is removed from a bag only if it is contained in that bag. (If this condition is satisfied then every $+LM$ and $-m$ cancel against a left-neighboring $-M$ and $+rm$.) For example, in order to show that this condition is met for $b := b + LM - M + rm - m$ we prove that M is in $b + LM$, m is in $b + rm$, and $M \neq m$. We are not in the simple situation that we can show that M is in b instead of in $b + LM$: the element M that is being removed from the bag is sometimes the element LM that has just been added. Notice that the order of the bag operations is important: $b := b + LM - M + rm - m$ and $b := b + LM - m + rm - M$ are not equivalent. We prove

- (a) $LM > m \ \wedge \ M > rm \ \Rightarrow \ M \in b + LM \ \wedge \ m \in b + rm \ \wedge \ M \neq m$
- (b) $LM > m \ \wedge \ M \leq rm \ \Rightarrow \ m \in b + LM$
- (c) $LM \leq m \ \wedge \ M > rm \ \Rightarrow \ M \in b + rm$.

case (a)

$$
\begin{array}{ll}
 & M \in b + LM \ \wedge \ m \in b + rm \ \wedge \ M \neq m \\
= & \quad \{ \ P \ \} \\
 & \max(lb + b) \in b + \max(lb) \ \wedge \ \min(b + rb) \in b + \min(rb) \\
 & \ \wedge \ \max(lb + b) \neq \min(b + rb) \\
\Leftarrow & \quad \{ \ \max(lb + b) = \max(b + \max(lb)), \ \min(b + rb) = \min(b + \min(rb)) \ \} \\
 & \max(lb + b) > \min(b + rb) \\
\Leftarrow & \quad \{ \ \max(lb + b) \geq \max(lb), \ P \ \} \\
 & LM > m
\end{array}
$$

case (b)

$$
\begin{array}{ll}
 & m \in b + LM \\
= & \quad \{\ P\ \} \\
 & \min(b + rb) \in b + \max(lb) \\
\Leftarrow & \quad \{\ \min(b) \in b\ \} \\
 & \min(b + rb) = \min(b) \\
\Leftarrow & \\
 & \max(b) \leq \min(rb) \\
\Leftarrow & \quad \{\ \max(b) \leq \max(lb + b),\ P\ \} \\
 & M \leq rm
\end{array}
$$

case (c) is similar to case b.

Termination of the algorithm follows directly from the observation that, in every step of the iteration, the number of inversions is decreased. (An inversion is a pair of elements from two different bags, where the left element exceeds the right element.) The number of inversions is a natural number and, hence, bounded from below which implies termination. Upon termination we have a state that satisfies both the invariant and the negation of all three guards. We, therefore, have

$$
\begin{array}{ll}
 & P \ \wedge \ LM \leq m \ \wedge \ M \leq rm \\
\Rightarrow & \\
 & \max(lb) \leq \min(b + rb) \ \wedge \ \max(lb + b) \leq \min(rb)
\end{array}
$$

upon termination, which is the required postcondition.

Notice that the algorithm is not correct if the last two guards are weakened to $LM > m$ and $M > rm$ respectively. It is then possible for elements to be removed from a bag of which they are not an element, implying that the union of all bags is not constant.

Statement $M := \max(b + LM)$ does not change M in the second guarded command, and may, therefore, be omitted. Similarly for $m := \min(b + rm)$ in the third guarded command.

1.5 A more efficient solution

The invariant proposed in the previous section was easy to guess (and understand), and led to a simple program. On closer inspection, however, it turns out that the program is not very efficient. Each step of the loop contains a construct for propagating maxima from left to right, and minima

from right to left. This propagation requires time proportional to the number of bags, making the execution time of the whole program quadratic in the number of bags. Operationally speaking, the processes are suspended most of the time on communications of global extremes. It seems to be more attractive to perform some exchanges of local extremes between neighbors in the mean time: we may hope to obtain a program whose execution time is linear in the number of bags instead of quadratic. This idea is not easily translated into a program, mainly because detecting the end of "the mean time" is nontrivial. A similar effect, however, can be obtained in a different way. Exchanges of local extremes between neighbors may be performed while, in passing, global extremes are computed. The global extremes can be computed by some sort of approximation technique. Formally, this amounts to weakening the invariant from $LM = \max(lb)$ to $LM \geq \max(lb)$, and $rm = \min(rb)$ to $rm \leq \min(rb)$. If we stick to the terms $M = \max(b + LM)$ and $m = \min(b + rm)$, as well as the other terms, then the conjunction of $LM \leq m$ and $M \leq rm$ and the invariant implies the postcondition. Hence, the weaker invariant is still sufficiently strong.

If we aim at a program whose structure is similar to the program in the previous section, we have a loop in which each step corresponds to a communication with the left neighbor, or with the right neighbor, or both. Deadlock is avoided if neighbors initiate their mutual communications under the same condition. Since only approximations of global extremes are locally available, we cannot simply use $LM > m$ and $M > rm$. Since LM is obtained from the, previously communicated, left neighbor's M, the left neighbor is to initiate a communication based on the previous value of M, say PM. This leads to invariant Q and to the program of Figure 1.5.

$$Q: \quad LM \geq \max(lb) \ \wedge \ PM \geq M = \max(b + LM) \ \wedge \ rm \leq \min(rb) \ \wedge$$

$$pm \leq m = \min(b + rm) \ \wedge \ \#b = \#B \ \wedge \ lb + b + rb = lB + B + rB$$

Notice that, due to the exchange of local extremes between neighbors rather than the propagation of global extremes, it may be necessary to replace two elements from the local bag. Hence, this algorithm is applicable only to the case in which each bag (except for the leftmost and rightmost bags) contains at least two elements.

We prove the correctness of this algorithm. Consider two processes that are neighbors in the linear array. We show that PM and rm in the left process have the same value as LM and pm in the right process. Initially we have $PM = +\infty$ and $rm = -\infty$ in the left process (since its rb is nonempty). Similarly we have $LM = +\infty$ and $pm = -\infty$ in the right process. The four variables are assigned a new value only when the two processes communicate with each other. The relevant statements are

$$r?(y, rm) \parallel R!(\max(b), M); \ PM := M$$

$$
\begin{array}{l}
\textbf{if } lb = \emptyset \rightarrow LM := -\infty \ [\!] \ lb \neq \emptyset \rightarrow LM := +\infty \textbf{ fi};\\
\textbf{if } rb = \emptyset \rightarrow rm := +\infty \ [\!] \ rb \neq \emptyset \rightarrow rm := -\infty \textbf{ fi};\\
M, PM, m, pm := \max(b + LM), +\infty, \min(b + rm), -\infty;\\
\textbf{do } LM > pm \ \wedge \ PM > rm \rightarrow\\
\qquad L?(x, LM) \parallel l!(\min(b), m) \parallel r?(y, rm) \parallel R!(max(b), M);\\
\qquad \textbf{if } x > \min(b) \ \wedge \ \max(b) > y \rightarrow b := b - \min(b) - \max(b) + x + y\\
\qquad\ [\!] \ x > \min(b) \ \wedge \ \max(b) \leq y \rightarrow b := b - \min(b) + x\\
\qquad\ [\!] \ x \leq \min(b) \ \wedge \ \max(b) > y \rightarrow b := b - \max(b) + y\\
\qquad\ [\!] \ x \leq \min(b) \ \wedge \ \max(b) \leq y \rightarrow skip\\
\qquad \textbf{fi};\\
\qquad M, PM, m, pm := \max(b + LM), M, \min(b + rm), m\\
[\!] \ LM > pm \ \wedge \ PM \leq rm \rightarrow\\
\qquad L?(x, LM) \parallel l!(\min(b), m);\\
\qquad \textbf{if } x > \min(b) \rightarrow b := b - \min(b) + x\\
\qquad\ [\!] \ x \leq \min(b) \rightarrow skip\\
\qquad \textbf{fi};\\
\qquad M, m, pm := \max(b + LM), \min(b + rm), m\\
[\!] \ LM \leq pm \ \wedge \ PM > rm \rightarrow\\
\qquad r?(y, rm) \parallel R!(max(b), M);\\
\qquad \textbf{if } \max(b) > y \rightarrow b := b - \max(b) + y\\
\qquad\ [\!] \ \max(b) \leq y \rightarrow skip\\
\qquad \textbf{fi};\\
\qquad M, PM, m := \max(b + LM), M, \min(b + rm)\\
\textbf{od}
\end{array}
$$

FIGURE 1.5.

in the left process, and

$$L?(x, LM) \parallel l!(\min(b), m);\ \ pm := m$$

in the right process. Inspection reveals that both PM and LM are assigned the value M, and that both rm and pm are assigned the value m. Hence, the correspondence between the variables is maintained. Consequently, the two processes initiate their mutual communications under the same condition, which excludes deadlock.

Next we show that the operations on b do not falsify the invariant. Inspection of the communication statements (as in the paragraph above) reveals that $\max(b)$ and y in the left process correspond to x and $\min(b)$ in the right process. Hence the updates of the bags are performed under the same condition and change neither the union of the bags nor the size of each bag. Notice that the assumption $\#b \geq 2$ is essential here.

In the same vein the invariance of $LM \geq \max(lb)$ and $PM \geq M = \max(b + LM)$ may be proved.

It remains to prove termination. To that end we strengthen the invariant to express that M is a very good approximation of $\max(lb + b)$. In fact, we have either $M = +\infty$ or $M = \max(lb + b)$. We can even prove that also $M = \max(b)$ holds in the latter case. This expresses the (strong) property that the largest value of $lb + b$ resides in bag b, and that the second largest value of $lb + b$ resides either in b or in the left-neighbor's bag, etc.. Furthermore, we show that $M = +\infty$ does not persist too long. More precisely we show that, in the process which has k other processes to its left, $M = \max(b) = \max(lb + b)$ holds after k iterations of the loop. We postulate that

$$LM = PM = M = +\infty \ \vee \ (+\infty > LM \geq \max(lb)$$

$$\wedge \ PM \geq M = \max(b) = \max(lb + b))$$

is an invariant, and we verify this claim. Initially $M = +\infty$ holds in every process except in the leftmost process ($k = 0$) in which $M = \max(b) = \max(lb + b)$. If $M = +\infty$ then the process initiates a communication to the left (since $M = +\infty$ implies $LM = +\infty$, and $pm < +\infty$). The relevant statements are

$$L?(x, LM) \cdots;$$
$$\textbf{if } x > \min(b) \cdots \rightarrow b := b - \min(b) + x \cdots \textbf{ fi};$$
$$M := \max(b + LM)$$

together with

$$R!(\max(b), M)$$

in the left process. If $M = +\infty$ holds in the left process prior to this step, then $LM = M = +\infty$ holds in the right process after this step. If $M = \max(b) = \max(lb + b)$ holds in the left process prior to this step, then the statement $L?(x, LM)$ in the right process leads to $x = LM = \max(lb)$. Hence, the updates of b and M lead to $M = \max(b) = \max(lb + b)$ in the right process, one iteration after this relation has been established in its left neighbor. Notice that the update of the bag in the left neighbor process may falsify $LM = \max(lb)$, but $LM \geq \max(lb)$ is maintained. As a result, in each process we have $M = \max(b) = \max(lb + b)$ after a number of steps equal to the number of processes. Similarly, $m = \min(b) = \min(b + rb)$ holds. When this state has been reached it is not guaranteed that the variant function from the previous two sections is decreased with every iteration of the loop. That variant function contained the bags only, and it is possible that no bag is changed by an iteration of the loop. However, if in this state the bag is not changed then it is the last iteration of the loop: if, for example, $LM > pm$ and $x \leq \min(b)$ then $LM = x \leq \min(b) = m$, and pm is set to m, thereby falsifying $LM > pm$ which excludes further iterations containing a communication to the left.

Upon termination we have the invariant and the negation of the guards

$$\begin{array}{l} \quad Q \ \wedge \ LM \leq pm \ \wedge \ PM \leq rm \\ \Rightarrow \\ \quad \max(lb) \leq LM \leq pm \leq m = \min(b + rm) \ \wedge \\ \quad \max(b + LM) = M \leq PM \leq rm \leq \min(b) \\ \Rightarrow \\ \quad \max(lb) \leq \min(b + rb) \ \wedge \ \max(lb + b) \leq \min(rb) \end{array}$$

which is the required postcondition.

The time complexity of the present solution is linear in the number of bags, N say. Thus, we have gained a factor of N at the expense of sending two integers per communication instead of one, and the addition of two integer variables per process. If each bag contains k elements, the number of iterations is $N \cdot k$ in the worst case. Assuming that the operations on a bag are $O(\log(k))$ each, this implies that the worst case time complexity is $O(N \cdot k \cdot \log(k))$.

In this program the guards of the second and third alternative of the loop may be weakened to $LM > pm$ and $PM < rm$ respectively, without falsifying the invariant. It has the advantage that the program may be simplified (by omitting the first alternative) and that the requirement $\#b \geq$

2 may be weakened to $\#b \geq 1$, but it has the distinct disadvantage that the program does not necessarily terminate: if both guards are true it is possible that selection of one of the alternatives does not change the state in either of the two processes involved. If fair selection of the alternatives is postulated, then one can show that the variant function decreases eventually, which implies that the program terminates eventually.

1.6 Conclusion

We have presented this paper as an exercise in deriving parallel programs. First, a sequential solution to the problem is presented which is subsequently transformed into a parallel solution. Next, extra variables and communication channels are introduced. Finally, the invariant is weakened. The transformation steps are not automatic in the sense that absence of deadlock had to be proved separately.

The resulting algorithms have some of the flavor of Odd-Even transposition sort. They are, however, essentially different in two respects. In every step of the loop in Odd-Even transposition sort, a process communicates with only one of its two neighbors, whereas in evry step of the loop of our algorithms a process communicates with both its neighbors (as long as necessary). The other difference is that our algorithms are "smooth" (cf. [2]) in the sense that the execution time is much less for almost-sorted arrays than for hardly-sorted arrays, with a smooth transition from one to the other behavior. This is due to the conditions under which processes engage in communications.

Acknowledgements: We are grateful to Johan J. Lukkien for many discussions during the design of the algorithms, and to Wayne Luk for helpful remarks on the presentation.

References

[1] E.W. Dijkstra, *A Discipline of Programming*, (Prentice Hall, Englewood Cliffs, NJ 1976).

[2] E.W. Dijkstra, Smoothsort, an alternative for sorting in situ, *Science of Computer Programming* 1, (1982) 223-233.

[3] C.A.R. Hoare, Communicating Sequential Processes, *Comm. ACM* (1978) 666-677.

[4] C.A.R. Hoare, *Communicating Sequential Processes*, (Prentice-Hall International Series in Computer Science, 1985)

[5] A.J. Martin, An axiomatic Definition of Synchronization Primitives, *Acta Informatica* 16, (1981) 219-235.

[6] S. Owicki, D. Gries, An Axiomatic Proof Technique for Parallel programs, *Acta Informatica* 6, (1976) 319-340.

2

A Distributed Implementation of a Task Pool

- Published previously in *Research Directions in High-Level Programming Languages*, J.B. Banâtre and D. Le Métayer (Eds.), LNCS 574 (1992) 338-348

H. Peter Hofstee, Johan. J. Lukkien,and Jan L. A. van de Snepscheut

ABSTRACT In this paper we present a distributed algorithm to implement a task pool. The algorithm can be used to implement a processor farm, i.e., a collection of processes that consume tasks from the task pool and possibly produce tasks into it. There are no restrictions on which process consumes which task nor on the order in which tasks are processed. The algorithm takes care of the distribution of the tasks over the processes and ensures load balancing. We derive the algorithm by transforming a sequential algorithm into a distributed one. The transformation is guided by the distribution of the data over processes. First we discuss the case of two processes, and then the general case of one or more processes.

Keywords: load balancing, processor farm, distributed computation.

2.1 Program notation

For the sequential part of the algorithms, we use Edsger W. Dijkstra's guarded command language [1]. For statements $S0$ and $S1$, statement $S0||S1$ denotes their concurrent execution. The constituents $S0$ and $S1$ are then called processes. The statements may share variables (cf. [5]). We transform our algorithms in such a way, however, that the final code contains no shared variables and all synchronization and communication is performed by message passing. The semantics of the message passing primitives is as described in [4]. The main difference with C.A.R. Hoare's proposal in [2] is in the naming of channels rather than processes. In [3], the same author proposes to name channels instead of processes in communication commands, but differs from our notation by using one name per channel instead of our two: output command $R!E$ in one process is paired with input command $L?v$ in another process by declaring the pair (R, L) to be a channel between the two processes. Each channel is between two processes only. When declaring (R, L) to be a channel, we write the name

on which the output actions are performed first and the name on which the input actions are performed last.

For an arbitrary command A, let $\mathbf{c}\,A$ denote the number of completed A actions, i.e., the number of times that command A has been executed since initiation of the program's execution. The *synchronization requirement* (cf. [4]) fulfilled by a channel (R, L) is that

$$\mathbf{c}\,R = \mathbf{c}\,L$$

holds at any point in the computation.

The execution of a command results either in the completion of the action or in its suspension when its completion would violate the synchronization requirement. From suspension until completion an action is *pending* and the process executing the action is delayed. We introduce boolean $\mathbf{q}\,A$ equal to the predicate "an A action is pending". Associated with a channel (R, L) we use $\overline{R} \equiv \mathbf{q}R!$ and $\overline{L} \equiv \mathbf{q}L?$ as defined in [7]. $\overline{L}$, pronounced *probe of* L, evaluates to *true* if an $R!$ action is pending, and to *false* otherwise. The *progress requirement* states that actions are suspended only if their completion would violate the synchronization requirement, i.e., channel (R, L) satisfies

$$\neg\mathbf{q}\,R \;\vee\; \neg\mathbf{q}\,L \qquad .$$

The nth R action is said to match the nth L action. The completion of a matching pair of actions is called a *communication*. The *communication requirement* states that execution of matching actions $R!E$ and $L?v$ amounts to the assignment $v := E$. The semantics of the **if** statement is reinterpreted (cf. [2]). If all guards in an **if** statement evaluate to *false*, its execution results in suspension of the execution rather than **abort**. When the guards contain no shared variables, an if statement that is suspended remains suspended and therefore this definition is compatible with the original semantics.

2.2 Statement of the problem

Given is a finite, constant number of processes $\mathbf{C}_{(0\leq i<N)}$ capable of consuming or producing tasks. Associated with each of these processes is a pool process $\mathbf{P}_i$. Each process $\mathbf{C}_i$ communicates only with its pool process. The pool processes also communicate with one another. Each pool can hold an unbounded number of tasks. (We will address bounded pool sizes in section 2.5.) The problem is to construct pool processes in such a way that no process $\mathbf{C}_i$ is idle forever if there is enough work to be done. We do not want to impose restrictions on the processes $\mathbf{C}_i$. Since a process $\mathbf{C}_i$ might, once activated, produce another task at any time, we cannot expect the algorithm for the pool to terminate. The requirements for the program can be formulated as follows:

- (0). A situation in which a process is idle whereas there are enough tasks in the network to keep all processes busy does not persist.
- (1). If the processes $\mathbf{C}_i$ do not consume or produce tasks, the number of communications is bounded.

The first requirement guarantees progress, whereas the second requirement guarantees quiescence in the absence of communications between the $\mathbf{C}_i$ and $\mathbf{P}_i$.
Coming up with the proper formalization of the first condition is nontrivial. On the one hand the condition should not be overly restrictive, since this will give rise to a large communication overhead. On the other hand several candidates for this condition had to be rejected because they allowed stable states that were undesirable. We finally replaced the first condition with the following:

- (0)'. Either communications are pending or $(\forall i :: p_i \leq T) \ \vee \ (\forall i :: p_i \geq T)$.
- (0)". There is no deadlock.

T is some fixed positive number, the threshold, and p_i denotes the number of elements in pool i. If we assume items are communicated one at a time, the solution seems obvious; select a pair of pools where one has more than T and the other has less than T elements and send an element from the fuller to the emptier pool. Written as a global sequential program, and ignoring the processes $\mathbf{C}_i$ and the actual communication of tasks, we obtain the program in Figure 2.1. The quantified expression $(\llbracket i :: \mathit{guarded\text{-}command}_i)$ should be interpreted as one guarded command for every i.

$$\textbf{do} \ (\llbracket \ i, j :: p_i < T \ \ \wedge \ \ p_j > T \rightarrow p_i, p_j := p_i + 1, p_j - 1 \ ) \ \textbf{od}$$

FIGURE 2.1.

The negation of the guards is $(\forall i :: p_i \leq T) \ \vee \ (\forall i :: p_i \geq T)$. Termination is guaranteed by the bound function $\sum_i |p_i - T|$. This sequential algorithm can be transformed into a distributed algorithm on a fully connected graph. However, since the resulting algorithm would not scale well with the size of the graph, we do not pursue this solution here.

We have tried to solve this problem by an approach similar to that in [6], that is, by examining a linear communication network for the processes $\mathbf{P}_i$ and by introducing variables that will approach $(\exists j : 0 \leq j < i : p_j < T)$ etc.. This gives rise to an extremely complicated algorithm if we do not want to make assumptions about the $\mathbf{C}_i$, mainly because any global information is volatile.

2.3 An asymmetric solution

If the process graph is not fully connected, the problem cannot be solved in a symmetric way on the basis of information about a process and its neighbors only. To see this, consider the following series of values for the pools:

$$p_0 = T - 1 \qquad p_1 = T \qquad p_2 = T + 1$$

A symmetric bound function such as $\sum_i |p_i - T|$ would not be decreased by communicating an item from pool $\mathbf{P}_2$ to pool $\mathbf{P}_1$ or from pool $\mathbf{P}_1$ to pool $\mathbf{P}_0$. Stated differently, a solution that would do this cannot be expected to satisfy our second condition: we cannot put an upperbound on the number of communications.
The solution is as follows. We break the symmetry by constructing a rooted, directed tree. We define $p\text{-}edge_{ij}$ to be true just when an edge from i to j directed towards the root exists in the tree. We define w_i to be the distance in the tree of node i from the root. We ensure progress by building our solution on the following bound function:

$$\sum_i |p_i - T| \cdot w_i$$

We have now solved the problem of the 'equalities' in our previous example. After assigning weight i to pool $\mathbf{P}_i$ we now have, for example, that communicating a task from $\mathbf{P}_2$ to $\mathbf{P}_1$ decreases the bound function, whereas a communication from $\mathbf{P}_1$ to $\mathbf{P}_0$ increases the bound function. The global effect is that 'equalities' will tend to accumulate from the leaves on up. Our solution, written once again as a global sequential program is:

do $(\llbracket i,j : p\text{-}edge_{ij} : p_i > T \ \wedge \ p_j \leq T \rightarrow p_i, p_j := p_i - 1, p_j + 1)$
$\llbracket \ (\llbracket i,j : p\text{-}edge_{ij} : p_i < T \ \wedge \ p_j \geq T \rightarrow p_i, p_j := p_i + 1, p_j - 1)$
od

FIGURE 2.2.

Upon termination, $p_i > T$ in any node, implies $p_{root} > T$, as can easily be proven by induction on the length of a path along *p-edges* from i to the root. Similarly, $p_i < T$ in any node implies $p_{root} < T$. Upon termination we have, therefore $\neg(\exists i,j :: p_i < T \ \wedge \ p_j > T)$ which is equivalent to $(\forall i :: p_i \leq T) \ \vee \ (\forall i :: p_i \geq T)$.

2.4 Program transformation

We now transform the global sequential solution into a local and distributed one. We limit ourselves to two processes for now. Process $\mathbf{P}_0$ is the designated root. The only edge in the tree is from $\mathbf{P}_1$ to $\mathbf{P}_0$. In this case our sequential solution simplifies to Figure 2.3. It describes the combined effect of $\mathbf{P}_0$ and $\mathbf{P}_1$.

$$
\begin{array}{l}
\textbf{do } p_1 > T \ \wedge \ p_0 \leq T \rightarrow p_1, p_0 := p_1 - 1, p_0 + 1 \\
\ [] \ p_1 < T \ \wedge \ p_0 \geq T \rightarrow p_1, p_0 := p_1 + 1, p_0 - 1 \\
\textbf{od}
\end{array}
$$

FIGURE 2.3.

Next, we split it into two processes. We introduce new variables q_i to hold the value of the p_i of the other process. Using channels (OUT_0, IN_1), and (OUT_1, IN_0) to communicate tasks, we obtain the program in Figure 2.4 in which we have eliminated all shared variables.

$$
\begin{array}{ll}
\mathbf{P}_0 \equiv & \{\textit{initialization establishes } q_1 = p_1\} \\
& \textbf{do } q_1 > T \ \wedge \ p_0 \leq T \rightarrow IN_1?task;\ q_1, p_0 := q_1 - 1, p_0 + 1 \\
& \ [] \ q_1 < T \ \wedge \ p_0 \geq T \rightarrow OUT_1!task;\ q_1, p_0 := q_1 + 1, p_0 - 1 \\
& \textbf{od}
\end{array}
$$

$$
\begin{array}{ll}
\mathbf{P}_1 \equiv & \{\textit{initialization establishes } q_0 = p_0\} \\
& \textbf{do } p_1 > T \ \wedge \ q_0 \leq T \rightarrow OUT_0!task;\ p_1, q_0 := p_1 - 1, q_0 + 1 \\
& \ [] \ p_1 < T \ \wedge \ q_0 \geq T \rightarrow IN_0?task;\ p_1, q_0 := p_1 + 1, q_0 - 1 \\
& \textbf{od}
\end{array}
$$

FIGURE 2.4.

The invariant $p_0 = q_0 \ \wedge \ p_1 = q_1$ guarantees that communications are started under the same condition in both processes. Statement $OUT_1!task$ selects an arbitrary task from the part of the pool stored in $\mathbf{P}_0$ and transmits it via channel OUT_1. Since guard $p_0 \geq T$ holds, the local task pool is nonempty. Statement $IN_1?task$ receives a task via channel IN_1 and adds

it to the local part of the task pool. We leave the rest of the verification of this program to the reader.
The next transformation includes the processes $\mathbf{C}_i$. We simulate their effect by including the line

$$\overline{P_i} \rightarrow P_i?p_i$$

where the channel $(\ldots, P_i)$ connects each pool process to its computing process. The invariant $p_0 = q_0 \ \wedge \ p_1 = q_1$ of the previous program can now no longer be maintained without extra communication. To keep this extra communication to a minimum, we keep track of the last communicated values of the p_i in the variables op_i and start a new communication only if a change in p_i affects one of the guards in its neighboring pool process. We cannot predict when another communication may be started by the other process, therefore the communication channels have to be probed. After a communication has started, we must first update the values of the q_i to ensure that communicating a task will indeed result in progress, and to ensure that communications match. This suggests the program shown in Figure 2.5. To simplify the initialization we start with an empty task pool.
We prove the generalization of this algorithm to the tree described in section 2.3. The program can be found in Figure 2.6. The channels are (OUT_{ij}, IN_{ji}), for which $p\text{-}edge_{ij} \vee p\text{-}edge_{ji}$. For those i and j we have introduced variables op_{ij} to keep track of the value of p_i last communicated from process $\mathbf{P}_i$ to process $\mathbf{P}_j$ and variables q_{ij} to hold the values of the p_j last communicated by process $\mathbf{P}_j$.
We need the following invariant: $(\forall i, j :: q_{ij} = op_{ji})$. The invariant holds initially and continues to hold because the statement
$(OUT_{ij}!p_i \parallel IN_{ij}?q_{ij})$; $op_{ij} := p_i$ in process $\mathbf{P}_i$ pairs with statement
$(OUT_{ji}!p_j \parallel IN_{ji}?q_{ji})$; $op_{ji} := p_j$ in process $\mathbf{P}_j$ to yield the assignment
$op_{ij}, q_{ji}, op_{ji}, q_{ij} := p_i, p_i, p_j, p_j$ which clearly maintains the invariant. Furthermore assignment $q_{ij}, p_i, op_{ij} := q_{ij} - 1, p_i + 1, op_{ij} + 1$ in the first alternative of the first innermost **do** construct of process $\mathbf{P}_i$ is guaranteed to match with assignment $p_i, q_{ij}, op_{ij} := p_i - 1, q_{ij} + 1, op_{ij} - 1$ in the second innermost **do** construct in process $\mathbf{P}_j$ because the preceding statements establish $p_i = q_{ji} \ \wedge \ p_j = q_{ij}$. The other alternative is similar and both pairs maintain the invariant. Finally, statement $P_i?p_i$ does not change q_{ij} or op_{ij}.
We check our three requirements one by one:

- (0)'. Either there are communications pending or
  $(\forall i :: p_i \leq T) \ \vee \ (\forall i :: p_i \geq T)$.

We assume all processes have been suspended on the guards in the **if** statements and that $(\exists i, j :: p_i < T \ \wedge \ p_j > T)$ and derive a contradiction as in section 2.3. If $p_i < T$ and all guards are false, then $(\forall j : p\text{-}edge_{ij} : q_{ij} < T)$. Unless i is the root, such a j exists. The invariant implies $op_{ji} < T$ for that j and the falsity of the guard $sign(p_j - T) \neq sign(op_{ji} - T)$ implies $p_j < T$.

$\mathbf{P}_0 \equiv p_0, q_1, op_0 := 0, 0, 0;$
do $true \rightarrow$
if $\overline{IN_1} \vee$
$(q_1 > T \ \wedge \ p_0 \leq T) \vee (q_1 < T \ \wedge \ p_0 \geq T) \vee$
$(sign(p_0 - T) \neq sign(op_0 - T)) \rightarrow$
$(OUT_1!p_0 \parallel IN_1?q_1); \ op_0 := p_0;$
do $q_1 > T \wedge p_0 \leq T \rightarrow$
$IN_1?task; \ q_1, p_0, op_0 := q_1 - 1, p_0 + 1, op_0 + 1$
$[\!]$ $q_1 < T \wedge p_0 \geq T \rightarrow$
$OUT_1!task; \ q_1, p_0, op_0 := q_1 + 1, p_0 - 1, op_0 - 1$
od
$[\!]$ $\overline{P_0} \ \rightarrow \ P_0?p_0$
fi
od

$\mathbf{P}_1 \equiv q_0, p_1, op_1 := 0, 0, 0;$
do $true \rightarrow$
if $\overline{IN_0} \vee$
$(p_1 > T \ \wedge \ q_0 \leq T) \vee (p_1 < T \ \wedge \ q_0 \geq T) \vee$
$(sign(p_1 - T) \neq sign(op_1 - T)) \rightarrow$
$(OUT_0!p_1 \parallel IN_0?q_0); \ op_1 := p_1;$
do $p_1 > T \wedge q_0 \leq T \rightarrow$
$OUT_0!task; \ p_1, q_0, op_1 := p_1 - 1, q_0 + 1, op_1 - 1$
$[\!]$ $p_1 < T \wedge q_0 \geq T \rightarrow$
$IN_0?task; \ p_1, q_0, op_1 := p_1 + 1, q_0 - 1, op_1 + 1$
od
$[\!]$ $\overline{P_1} \ \rightarrow \ P_1?p_1$
fi
od

FIGURE 2.5.

$\mathbf{P}_i \equiv p_i := 0;\ (;\ j : p\text{-}edge_{ij} \vee p\text{-}edge_{ji} : q_{ij}, op_{ij} := 0, 0);$

do $true \rightarrow$

if $(\llbracket j : p\text{-}edge_{ji} :$

$\overline{IN_{ij}} \vee$

$(q_{ij} > T \ \wedge \ p_i \leq T) \vee (q_{ij} < T \ \wedge \ p_i \geq T) \vee$

$(sign(p_i - T) \neq sign(op_{ij} - T)) \rightarrow$

$(OUT_{ij}!p_i \parallel IN_{ij}?q_{ij});\ op_{ij} := p_i;$

do $q_{ij} > T \wedge p_i \leq T \rightarrow$

$IN_{ij}?task;\ q_{ij}, p_i, op_{ij} := q_{ij} - 1, p_i + 1, op_{ij} + 1$

$\llbracket\ q_{ij} < T \wedge p_i \geq T \rightarrow$

$OUT_{ij}!task;\ q_{ij}, p_i, op_{ij} := q_{ij} + 1, p_i - 1, op_{ij} - 1$

od

$)$

$\llbracket\ (\llbracket j : p\text{-}edge_{ij} :$

$\overline{IN_{ij}} \vee$

$(p_i > T \ \wedge \ q_{ij} \leq T) \vee (p_i < T \ \wedge \ q_{ij} \geq T) \vee$

$(sign(p_i - T) \neq sign(op_{ij} - T)) \rightarrow$

$(OUT_{ij}!p_i \parallel IN_{ij}?q_{ij});\ op_{ij} := p_i;$

do $p_i > T \wedge q_{ij} \leq T \rightarrow$

$OUT_{ij}!task;\ p_i, q_{ij}, op_{ij} := p_i - 1, q_{ij} + 1, op_{ij} - 1$

$\llbracket\ p_i < T \wedge q_{ij} \geq T \rightarrow$

$IN_{ij}?task;\ p_i, q_{ij}, op_{ij} := p_i + 1, q_{ij} - 1, op_{ij} + 1$

od

$)$

$\llbracket\ \overline{P_i} \ \rightarrow \ P_i?p_i$

fi

od

FIGURE 2.6.

By induction $p_{root} < T$. By symmetry we also have $p_{root} > T$, which establishes the contradiction.

- (0)". There is no deadlock.

The graph we have constructed is a tree. Since this is an acyclic structure, and since the algorithm guarantees that two processes cannot deadlock when they are committed to communicating with one another, there is no cycle in the communication dependencies. Therefore there is no deadlock.

- (1). If the processes $\mathbf{C}_i$ do not consume or produce tasks, the number of communications is bounded.

A bound function that decreases on every repetition of the outermost **do** construct (other than communications between a $\mathbf{P}_i$ and a $\mathbf{C}_i$) is

$$\sum_i |p_i - T| \cdot w_i + (\mathbf{N}\, i, j : p{-}edge_{ij} \vee p{-}edge_{ji} : p_i \neq q_{ji})$$

The proof is as follows. An execution of the first alternative in process $\mathbf{P}_i$ always matches with the execution of the second alternative in some process $\mathbf{P}_j$ for which $p\text{-}edge_{ji}$. For any such pair the first sum can never increase due to the fact that the first pair of communications establishes $p_i = q_{ji} \ \wedge \ p_j = q_{ij}$. If before the communication either $(sign(p_i - T) \neq sign(op_{ij} - T))$ in $\mathbf{P}_i$ or $(sign(p_j - T) \neq sign(op_{ji} - T))$ in $\mathbf{P}_j$, then the assignments $op_{ij} := p_i$ and $op_{ji} := p_j$ in $\mathbf{P}_i$ and $\mathbf{P}_j$ respectively, in conjunction with the invariant $op_{ij} = q_{ji} \ \wedge \ op_{ji} = q_{ij}$, imply that the variant function decreases. If $(sign(p_i - T) = sign(op_{ij} - T))$ in $\mathbf{P}_i$ and $(sign(p_j - T) = sign(op_{ji} - T))$ in $\mathbf{P}_j$, but all guards in the innermost **do** statements evaluate to false, so that the first part of the variant function does not decrease, either q_{ij} in $\mathbf{P}_i$ or q_{ji} in $\mathbf{P}_j$ must have changed in the first pair of communications and the second term in the invariant has decreased.

Note. The algorithm in Figure 2.6 can be somewhat simplified by omitting $(p_i > T \ \wedge \ q_{ij} \leq T) \vee (p_i > T \ \wedge \ q_{ij} \geq T)$ from the guard of the second outermost alternative. This can be done because for any pair $\mathbf{P}_i, \mathbf{P}_j$ of processes that may communicate the following holds:

$$\begin{array}{l} (p_i > T \ \wedge \ q_{ij} \leq T) \vee (p_i < T \ \wedge \ q_{ij} \geq T) \\ \Rightarrow \\ (q_{ji} > T \ \wedge \ p_j \leq T) \vee (q_{ji} < T \ \wedge \ p_j \geq T) \vee \\ sign(p_i - T) \neq sign(op_{ij} - T) \vee \\ sign(p_j - T) \neq sign(op_{ji} - T) \end{array}$$

which can be proven by using the invariant $op_{ij} = q_{ji} \ \wedge \ op_{ji} = q_{ij}$.

2.5 More thresholds

If we want to put stronger bounds on how much the pools can be allowed to differ, we can add more thresholds in a straightforward manner. Condition $(\forall i :: p_i \leq T) \vee (\forall i :: p_i \geq T)$ is replaced by $(\forall k :: (\forall i :: p_i \leq T_k) \vee (\forall i :: p_i \geq T_k))$. The algorithm is given in Figure 2.7.
The proof that our first two requirements are still being met is nearly identical to the case with one threshold and is left to the reader. However, to guarantee that the number of communications, on the absence of communications with the computing processes, is bounded, we have to insist that the threshold values differ by at least two. A bound function is then given by the following two-tuple, and the ordering is lexicographic ordering.

$$(\sum_i p_i^2 + (Ni, j : p{-}edge_{ij} \vee p{-}edge_{ji} : p_i \neq q_{ji}), \sum_i (MINk :: |p_i - T_k| \cdot w_i))$$

Without proof we state that, as in the previous algorithm, on every communication between two processes $\mathbf{P}_i$ and $\mathbf{P}_j$, either a task is being communicated, or the second term in the first element of the two-tuple decreases. If a task is being communicated, and p_i and p_j differed originally by more than one, the sum over the squares decreases. Without loss of generality we can state $w_i < w_j$, therefore $p\text{-}edge_{ji} = true$. If p_i and p_j originally differed by exactly one, then it follows that originally $p_i = T_k$ for some k and $|p_j - T_k| = 1$ for that k. The communication of the task then reduces the second element of the two-tuple by $w_j - w_i$. The fact that the T_k differ by at least two, is needed here to guarantee that the k's for which the terms in the second element of the two tuple are minimal do not change.
We can greatly reduce the amount of computation needed to evaluate the guards in the algorithm in Figure 2.7 by maintaining variables tl_i and tg_i such that

$$\begin{aligned} &tl_i \leq p_i \leq tg_i \ \wedge \ ((\exists k :: tl_i = T_k) \vee tl_i = 0) \ \wedge \\ &((\exists k :: tg_i = T_k) \vee tg_i = poolsize) \ \wedge \\ &\neg(\exists k :: tl_i < T_k \leq p_i \vee p_i \leq T_k < tg_i) \end{aligned}$$

All existential quantifications can then be removed from the guards.

2.6 Finite pool sizes

If the size of the pools is fixed, it is possible that our algorithm attempts to add a task to a pool that has reached its capacity. Not much can be done

$\mathbf{P}_i \equiv p_i := 0;\ (;\ j : p\text{-}edge_{ij} \vee p\text{-}edge_{ji} : q_{ij}, op_{ij} := 0, 0);$
do $true \rightarrow$
if $([]j : p\text{-}edge_{ji} :$
$\overline{INij} \vee$
$(\exists k :: (q_{ij} > T_k \ \wedge \ p_i \leq T_k) \vee (q_{ij} < T_k \ \wedge \ p_i \geq T_k)) \vee$
$(\exists k :: sign(p_i - T_k) \neq sign(op_{ij} - T_k)) \rightarrow$
$(OUT_{ij}!p_i \parallel IN_{ij}?q_{ij});\ op_{ij} := p_i;$
do $(\exists k :: q_{ij} > T_k \wedge p_i \leq T_k) \rightarrow$
$IN_{ij}?task;\ q_{ij}, p_i, op_{ij} := q_{ij} - 1, p_i + 1, op_{ij} + 1$
$[]\ (\exists k :: q_{ij} < T_k \wedge p_i \geq T_k) \rightarrow$
$OUT_{ij}!task;\ q_{ij}, p_i, op_{ij} := q_{ij} + 1, p_i - 1, op_{ij} - 1$
od
)
$[]\ ([]j : p\text{-}edge_{ij} :$
$\overline{IN_{ij}} \vee$
$(\exists k :: (p_i > T_k \ \wedge \ q_{ij} \leq T_k) \vee (p_i < T_k \ \wedge \ q_{ij} \geq T_k)) \vee$
$(\exists k :: sign(p_i - T_k) \neq sign(op_{ij} - T_k)) \rightarrow$
$(OUT_{ij}!p_i \parallel IN_{ij}?q_{ij});\ op_{ij} := p_i;$
do $(\exists k :: p_i > T_k \wedge q_{ij} \leq T_k) \rightarrow$
$OUT_{ij}!task;\ p_i, q_{ij}, op_{ij} := p_i - 1, q_{ij} + 1, op_{ij} - 1$
$[]\ (\exists k :: p_i < T_k \wedge q_{ij} \geq T_k) \rightarrow$
$IN_{ij}?task;\ p_i, q_{ij}, op_{ij} := p_i + 1, q_{ij} - 1, op_{ij} + 1$
od
)
$[]\ \overline{P_i} \ \rightarrow \ P_i?p_i$
fi
od

FIGURE 2.7.

about the situation where all pools are full, but it would be problematic if the program would deadlock in a situation where pool space is available in another processor. Since a communication between two pool processes does not increase the size of the largest pool, a problem can only occur in the communications between the $\mathbf{C}_i$ and their $\mathbf{P}_i$. If we replace the single channel by two channels, one over which tasks are consumed, and one over which tasks are produced, and replace the guard for P_{prod} by $\overline{P}_{prod} \wedge p_i < poolsize$ and assume that tasks are produced one at a time, the problem is solved. We can ensure that computation power is used efficiently until all buffers fill up by choosing one of the thresholds to be $poolsize - 1$.

2.7 Discussion

This paper is an attempt to deal in a systematic way with a problem that in our experience occurs frequently. In the past when solving a distributed programming problem of this type, we would construct the $\mathbf{C}_i$ for that particular problem, think of a strategy that would provide a decent load balance, and construct $\mathbf{P}_i$ for that specific problem. The present solution seems to be more general.

Acknowledgements: We thank Nan Boden for helpful comments on the manuscript.

References

[1] E.W. Dijkstra, *A Discipline of Programming*, (Prentice Hall, Englewood Cliffs, NJ 1976).

[2] C.A.R. Hoare, Communicating Sequential Processes, *Comm. ACM* (1978) 666-677.

[3] C.A.R. Hoare, *Communicating Sequential Processes*, (Prentice-Hall International Series in Computer Science, 1985)

[4] A.J. Martin, An Axiomatic Definition of Synchronization Primitives, *Acta Informatica* 16, (1981) 219-235.

[5] S. Owicki, D. Gries, An Axiomatic Proof Technique for Parallel programs, *Acta Informatica* 6, (1976) 319-340.

[6] H. P. Hofstee, A. J. Martin, J. L. A. van de Snepscheut, Distributed Sorting, *Science of Computer Programming* 15 (1990) 119-133

[7] A.J. Martin, The Probe: An Addition to Communication Primitives, *Information Processing Letters* 20 (1985) 125-130

3

Distributing a class of sequential programs

- To appear in *Mathematics of Program Construction* 1992 Conf. Proceedings, Springer LNCS series

H. Peter Hofstee [1]

ABSTRACT A class of sequential programs is distributed through a series of program transformations. To construct a concurrent solution, a sequential solution is given first. A decision is made about the distribution of the variables and the sequential solution is transformed so that guards at the outermost level can be evaluated using variables that will be allocated to one process only. Next we introduce processes and communication. The resulting distributed algorithm does not terminate, but it will become quiescent, and in this state the original postcondition will hold. The distributed algorithm is highly nondeterministic and not network specific. A synchronization primitive, the nonblocking channel, is introduced, and used to generalize the first distributed solution to a larger class of networks.
We give two examples of problems that can be solved with this approach. First we show how a more general version of the load-balancing algorithm of [7] can be derived as an instance of this class. Next we instantiate our solution to arrive at an algorithm for distributed sorting. Finally we refine this solution to arrive at a terminating distributed sorting algorithm.

3.1 Introduction

The algorithms we present in this paper have been constructed with a specific architecture in mind; a message passing multicomputer with many small nodes. Unlike single instruction multiple data (SIMD) machines, synchronization is not implied, but has to be programmed. It is unlikely that all processors will perform the same task in synchrony; if they would, a SIMD machine would be more suitable for the task. One would like to develop algorithms that can be used in a wide variety of contexts. The algorithm should therefore not depend on detailed analysis of a particular network, or on information about other tasks performed by the processors. These considerations make the cost of synchronization hard to predict, therefore

[1] H.Peter Hofstee is supported by an IBM graduate fellowship.

synchronization is kept to a minimum in the algorithms developed here. It also seems unlikely that deterministic programs, whose execution does not depend on properties of the network, or the workload of the processors, are most efficient. The algorithms presented in chapter two and three are therefore highly nondeterministic.

We will make only weak assumptions about the communication network that connects the nodes. We assume that we can use information about the network, and perhaps about the complete programs to make our final solution more deterministic and perhaps more efficient. In our experience, this turns out to be a good choice. The algorithm for distributed sorting presented here allows us to make use of more connections in a network than previous versions that were more deterministic, but also restricted to a specific communication topology. This property has caused it to outperform the previous solutions, even on relatively small networks. For the load-balancing example we did not have a deterministic solution to compare it to, but because nondeterminism is even more inherent to this problem, we do not doubt that the choice was appropriate for this problem as well.

The remainder of this paper focuses on the construction, and even though the above gives direction to the derivation, we try to make the formalism work for us as much as possible.

3.2 Transformations of a Sequential Program

We use an extended subset of Edsger W. Dijkstra's guarded command language [1]. We allow any commutative infix operator as a quantifier in a quantified expression. The meaning of such an expression is the meaning of the expression after expanding the expression for each value in its range separated by the operator.

$\{\forall(i :: v_i = V_i)\}$
do $[\!]$ $(i, j : iRj : c(v_i, v_j) \rightarrow S(v_i, v_j)\ )$ **od**
$\{\ Q\ \}$

FIGURE 3.1.

Our task is to find a distributed implementation for the program in Figure 3. 1. In this program iRj describes a finite directed acyclic graph; iRj is *true* just when the graph has a directed edge from node i to node j. Condition $c(v_i, v_j)$ is understood not to modify any variables. Condition c and statement S operate on their arguments only. The type of the variables

v_i is left unspecified. We assume the program is correct, there exist an invariant P and a bound function bf that satisfy

- $\forall(i :: v_i = V_i) \;\Rightarrow\; P$
- $\forall(i, j : iRj :$
  $P \;\wedge\; c(v_i, v_j) \;\wedge\; bf = BF \;\Rightarrow\; \mathrm{wp}(S(v_i, v_j), P \;\wedge\; bf < BF))$
- $P \;\wedge\; \forall(i, j : iRj : \neg c(v_i, v_j)) \;\Rightarrow\; Q$
- $bf > constant$

We intend to allocate variable v_i to process i. For any additional variables that we introduce, we will honor the convention that the first subscript denotes the process to which the variable will be allocated. However, we delay the introduction of processes and communications as long as possible, since properties of sequential programs are easier to prove. As a first step we introduce variables v_{ij} to replace the variables v_j. This allows us to evaluate the guard locally. We could have tried to maintain the invariant $\forall(i, j : iRj : v_{ij} = v_j)$. Maintaining this invariant would require statements that involve variables of more than two processes, which might make the distributed solution inefficient. A solution that involves variables from two processes in each alternative only is given in Figure 3. 2.

$\{\forall(i :: v_i = V_i)\}$
do $[\!]\ (i, j : iRj : c(v_i, v_{ij}) \rightarrow$ **if** $c(v_i, v_j) \rightarrow S(v_i, v_j)$
$[\!]\ \neg c(v_i, v_j) \rightarrow v_{ij} := v_j$
fi
)
$[\!]\ \ [\!]\ (i, j : iRj : v_j \neq v_{ij} \rightarrow v_{ij} := v_j)$
od

FIGURE 3.2.

It is easily verified this program will establish the same postcondition as the original program. The bound function is now a two tuple, $(bf, N(i, j : iRj : v_{ij} \neq v_j))$. This bound function decreases (lexicographic ordering) on every iteration and is bounded from below, therefore the second program also terminates. Curiously enough the variables v_{ij} need not appear in the invariant: the invariant is the same as the invariant of the original program. Therefore their initial value need not be specified.

We now remark that the alternative that we have added on the outermost level of the program has guard $v_j \neq v_{ij}$ which cannot be evaluated locally. We solve this problem by introducing variables ov_{ji} for which we maintain the invariant $\forall(i,j : iRj : ov_{ji} = v_{ij})$. This invariant can be maintained through statements involving variables of two processes only. The resulting program can be found in Figure 3. 3.

$\{\forall(i :: v_i = V_i)\}$
$; (i,j : iRj : ov_{ji} := v_{ij}) ;$
do $[\!] (i,j : iRj : c(v_i, v_{ij}) \rightarrow$ **if** $c(v_i, v_j) \rightarrow S(v_i, v_j)$
$[\!] \neg c(v_i, v_j) \rightarrow v_{ij}, ov_{ji} := v_j, v_j$
fi $)$
$[\!] \quad [\!] (i,j : iRj : v_j \neq ov_{ji} \rightarrow v_{ij}, ov_{ji} := v_j, v_j)$
od

FIGURE 3.3.

In order to establish the new conjunct in the invariant we have been forced to make a choice for the initial value of the ov_{ji}. The bound function for the program in Figure 3.2 can also be used to show that the algorithm in Figure 3.3 will terminate. The extended invariant and the negation of all guards implies the original postcondition. Therefore this program is a correct refinement as well.

Since the statements in the alternatives in Figure 3.3 contain references to variables that are not local to one process, communication will have to be introduced at this point. When we inspect the program in Figure 3. 3, we see that for a pair of processes (i,j) that satisfy iRj a communication can be initiated both by process i and by process j. Obviously this might create deadlock; i may be committed to a communication with j to jointly execute the first alternative, whereas j is dedicated to a communication with i to jointly execute the second alternative. One solution to this problem is to merge both alternatives and use the same channels, which implies that the communication will succeed if both processes are dedicated to communicating with each other, irrespective of which process initiated the communication. The algorithm where both alternatives have been merged is given in Figure 3.4.

This algorithm also maintains the invariant $P \ \wedge \ \forall(i,j : iRj : ov_{ji} = v_{ij})$ and also decreases $(bf, N(i,j : iRj : v_{ij} \neq v_j))$ on every iteration. Because we have not changed the guards, the algorithm in Figure 3.4 is a correct refinement as well.

$\{\forall(i :: v_i = V_i)\}$
$; (i, j : iRj : ov_{ji} := v_{ij}) ;$
do $[\!] (i, j : iRj : c(v_i, v_{ij}) \rightarrow S_0(v_i, v_j);\ v_{ij}, ov_{ji} := v_j, v_j\ )$
$[\!] \quad [\!] (i, j : iRj : v_j \neq ov_{ji} \rightarrow S_0(v_i, v_j);\ v_{ij}, ov_{ji} := v_j, v_j\ )$
od

where $S_0(v_i, v_j) \equiv$ **if** $c(v_i, v_j) \rightarrow S(v_i, v_j)\ [\!]\ \neg c(v_i, v_j) \rightarrow$ **skip fi**

FIGURE 3.4.

3.3 Processes and Communication

Now that all guards of the loop can be evaluated locally, we are ready to introduce processes and communication. For two statements S and T, $S||T$ denotes their parallel execution. The semantics of the message passing primitives is as described by A.J. Martin in [4]. The main difference with C.A.R. Hoare's proposal in [2] is in the naming of channels rather than processes. In [3], the same author proposes to name channels instead of processes in communication commands, but differs from our notation by using one name per channel instead of our two: output command $R!E$ in one process is paired with input command $L?v$ in another process by declaring the pair (R, L) to be a channel between the two processes. Each channel is between two processes only. When declaring (R, L) to be a channel, we write the name on which the output actions are performed first and the name on which the input actions are performed last. We allow the channels to be probed (cf. [5]) on one side. In this paper the side of the channel that is probed is the side on which the input actions are performed. The semantics of the **if** statement is reinterpreted (cf. [2]). If all guards in an **if** statement evaluate to *false*, its execution results in suspension of the execution rather than **abort**. When the guards contain no shared variables, an **if** statement that is suspended remains suspended forever and therefore this definition is compatible with the original semantics.

Our next task is to split the statement S_0 into two statements, each involving variables of one process only. The construction is given in figure 5.

A natural choice for S_0' and S_0'' that satisfies the requirement would be to have them differ from S_0 only in their arguments.

The only statement in Figure 3.4 that remains to be distributed is the initialization of the variables ov_{ji}. The invariant merely requires $ov_{ji} = v_{ij}$,

$S_0(v_i, v_j);\ \ v_{ij}, ov_{ji} := v_j, v_j$

$=$ {new variables x_i and x_j,

require $x_i = v_j \ \wedge \ x_j = v_i \ \wedge \ i \neq j \Rightarrow$

$S_0'(v_i, x_i) \parallel S_0''(x_j, v_j) = \ S_0(v_i, v_j);\ \ x_i, x_j := v_j, v_i$}

$(x_i := v_j \qquad\qquad \parallel x_j := v_i)$;

$((\ S_0'(v_i, x_i);\ \ v_{ij} := x_i) \qquad \parallel (S_0''(x_j, v_j);\ \ ov_{ji} := v_j)\ )$

$=$ {implement first two assignments and synchronization as communications using channels $(down_{ij}, down_{ji})$ and (up_{ji}, up_{ij}) }

$(\ (up_{ij}?x_i \parallel down_{ij}!v_i)\ ;\ \ S_0'(v_i, x_i);\ \ v_{ij} := x_i\ )$

$\parallel (\ (up_{ji}!v_j \parallel down_{ji}?x_j)\ ;\ \ S_0''(x_j, v_j);\ \ ov_{ji} := v_j\ )$

FIGURE 3.5.

hence we can choose **any** initial value for these variables as long as it's the same for corresponding v_{ij} and ov_{ji}.

We have chosen not to worry about termination at this stage. One problem for which we have used this solution, a distributed load-balancing algorithm, was intended not to terminate. We arrive at the solution in Figure 3. 6. We have renamed the dummy in the first and fourth alternative in order to textually separate alternatives that result in a communication with a predecessor in R from alternatives that result in a communication with a successor.

The first two alternatives jointly implement the first alternative of Figure 3.4. Since the original guard is evaluated in one of the two processes only, a new guard has to be found for the other process that becomes *true* just when the first one is. Since the body of the alternative starts with communication actions, probing one of the channels gives us just the right condition. Following [5], $\overline{c}$ (read c-probe) evaluates to *true* just when a communication on c is pending.

We have replaced a terminating algorithm by a nonterminating one, and therefore cannot expect that this algorithm implements our original one. We do expect the following:

(1) When all processes are quiescent, and this condition is stable, the original postcondition is established.

(2) After finite time all processes are quiescent and stay quiescent.

Quiescence is defined as the state in which a process is suspended. We now show that the first requirement is **not** met by the program in Figure 3.6. The relation iRj does not allow directed cycles, but it does allow cy-

$\{\forall(i :: v_i = V_i)\}$
$\|(i ::$
$\quad ; (j : iRj : v_{ij} := V) ;$
$\quad ; (h : hRi : ov_{ih} := V) ;$
$\quad$ **do** $true \rightarrow$
$\quad\quad$ **if** $[\!]\ (j : iRj : c(v_i, v_{ij}) \rightarrow (up_{ij}?x_i \| down_{ij}!v_i) ;$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad S_0'(v_i, x_i);\ v_{ij} := x_i\ )$
$\quad\quad [\!]\ [\!]\ (h : hRi : \overline{down_{ih}} \rightarrow (up_{ih}!v_i \| down_{ih}?x_i) ;$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad S_0''(x_i, v_i);\ ov_{ih} := v_i\ )$
$\quad\quad [\!]\ [\!]\ (h : hRi : v_i \neq ov_{ih} \rightarrow (up_{ih}!v_i \| down_{ih}?x_i)\ ;\ )$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad S_0''(x_i, v_i);\ ov_{ih} := v_i\ )$
$\quad\quad [\!]\ [\!]\ (j : iRj : \overline{up_{ij}} \rightarrow (up_{ij}?x_i \| down_{ij}!v_i)$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad S_0'(v_i, x_i);\ v_{ij} := x_i\ )$
$\quad\quad$ **fi**
$\quad$ **od**
$)$

FIGURE 3.6.

cles. Since the communications may go in either direction any cycle may cause deadlock when all processes in such a cycle are suspended on a communication with a successor in the cycle. If the graph represented by iRj is cycle free, the above is a correct implementation of the original algorithm. Because we do not want to restrict R, we opt for a different solution. We do not give a proof of correctness of the algorithm in Figure 3.6 for restricted iRj, but we will prove correctness of the more general version.

3.4 Nonblocking Channels

We require that all communications that do not follow the direction of the graph terminate. This requires the introduction of a special kind of channel (cf. [7]) with the property that any input action on the channel is followed by an output action of the channel. We do **not** require, however, that the number of outputs matches the number of inputs, or make assumptions about the values the channel produces.

With, for a channel $c = (c_i, c_o)$,

- $\mathbf{r}c \equiv$ number of $c_i?$ actions that have not been followed by a $c_i!$ action

and, following [4],

- $\mathbf{c}$A $\equiv$ number of completed A-actions
- $\mathbf{q}$A $\equiv$ 'an A-action is pending'

a nonblocking channel is characterized by

- $\neg\mathbf{q}c_i!$
- $\mathbf{c}c_o? \leq \mathbf{c}c_i!$
- $0 \leq \mathbf{r}c \leq slack$
- $\mathbf{q}c_o? \Rightarrow \mathbf{r}c = 0$

In the above the nonblocking channel is represented by a pair (c_i, c_o) and two operations, $c_i!$ and $c_o?$, which satisfy the properties given. Another example of a construct that satisfies these axioms is a special kind of semaphore, represented by a single variable, with the usual V-operation that increases the semaphore, but a P-operation that suspends if the semaphore is 0, but sets it to 0 and completes if it is greater than 0. In this case $slack = 1$.

Figure 3.7 gives an implementation of a nonblocking channel with $slack = 2$ that does not share variables between processes. When the values communicated are irrelevant, they are omitted.

```
process nbchan(in, out) ≡
        b := false;
        do true →
            if ¬in → in?; b := true
            [] ¬into ∧ b → into?; b := false
            fi
        od
||
        do true → inti!; out! od
```

FIGURE 3.7.

Since both of the alternatives in the alternative statement of the first process are probed, and since both alternatives involve no further communications, both are guaranteed to terminate, hence no output action on the channel is ever suspended. The local boolean ensures that the number of communications on *out* never exceeds the number of communications on *in* by more than the slack of the local channel (*inti*, *into*), and ensures the first alternative in the first process is chosen eventually. We assume the slack is zero; since both processes of the nonblocking channel can reside on the same processor, this does not present complications when implementing the channel. We can get rid of this assumption by exchanging *inti*! and *into*? in the program, but we have not done so because some implementations of concurrent programming languages do not allow the input side of a channel to be probed. At most two output actions can follow an input action, hence the $slack = 2$ for this nonblocking channel. If the last action on the channel was an input action, either $b = true$ or the second process is suspended on an output action. In either case an output action will not suspend.

3.5 A Solution with Nonblocking Channels

The new channel allows us to modify the solution in Figure 3.6. For a pair of processes (i, j) for which iRj holds, the code for the alternatives that implement a communication initiated by process i remains unchanged. Process j however, uses a nonblocking channel to process i to request a joint action. The action by such a process j on the nonblocking channel does not lead

to progress. That problem is resolved using the boolean variables $sent_{ij}$. We use the additional channels (to_{ij}, to'_{ij}) and $(from_{ij}, from_{ji})$ to communicate with the nonblocking channel processes. Finally, we have changed the order of the statements in such a way that if alternatives that textually precede others are more likely to be executed, long chains of processes waiting for each other are less likely to occur. Correctness of this algorithm, and the following ones, does not depend on that order: the semantics of the **if** statements merely states that one of the alternatives for which the guard evaluates to *true* is executed. The resulting algorithm is given in Figure 3.8.

$$
\begin{array}{l}
\{\forall(i :: v_i = V_i)\} \\
\|(i :: \\
\quad ; (j : iRj : v_{ij} := V) \; ; \\
\quad ; (h : hRi : ov_{ih} := V \; ; \; sent_{ih} := false) \; ; \\
\quad \textbf{do } true \rightarrow \\
\qquad \textbf{if } [] \; (h : hRi : \overline{down_{ih}} \;\; \rightarrow \;\; sent_{ih} := false \; ; \\
\qquad\qquad\qquad\qquad\qquad\qquad (up_{ih}!v_i \| \; down_{ih}?x_i) \; ; \\
\qquad\qquad\qquad\qquad\qquad\qquad S_0''(x_i, v_i) \; ; \;\; ov_{ih} := v_i) \\
\qquad [] \; [] \; (j : iRj : c(v_i, v_{ij}) \;\; \rightarrow \;\; (up_{ij}?x_i \| \; down_{ij}!v_i) \; ; \\
\qquad\qquad\qquad\qquad\qquad\qquad S_0'(v_i, x_i) \; ; \;\; v_{ij} := x_i) \\
\qquad [] \; [] \; (j : iRj : \overline{from_{ij}} \rightarrow \qquad from_{ij}?; \;\; (up_{ij}?x_i \| \; down_{ij}!v_i) \; ; \\
\qquad\qquad\qquad\qquad\qquad\qquad S_0'(v_i, x_i) \; ; \;\; v_{ij} := x_i) \\
\qquad [] \; [] \; (h : hRi : v_i \neq ov_{ih} \;\; \wedge \;\; \neg sent_{ih} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad to_{ih}!; \;\; sent_{ih} := true \;) \\
\qquad \textbf{fi} \\
\quad \textbf{od} \\
\;) \\
\| \\
\|(i, j : jRi : nbchan(to'_{ij}, from_{ij}))
\end{array}
$$

FIGURE 3.8.

We prove the program in Figure 3.8 satisfies the following properties:
(1) When the program is quiescent, the original postcondition is estab-

lished.

(2) The program becomes quiescent in finite time.

Proof of (1):

By definition, quiescence occurs when all processes are suspended. Processes can be suspended either because all guards in the **if** statement are *false* or because a process is suspended on a communication. We show that if any process is suspended on a communication, not all processes are suspended. A process cannot be suspended on any of the communications in the first alternative; $\overline{down_{ij}}$ becomes *true* only when another process is committed to a set of communications matching those in the first alternative. A process cannot be suspended on the communication in the fourth alternative either because it is a communication on a non-blocking channel. Thus a process i can be suspended only on one of the communications in the second or third alternatives, i.e. a communication with a process j such that iRj holds. It follows that process j must be suspended on a communication, since at least one of the guards in its alternative statement ($\overline{down_{ij}}$) is true. From the finiteness of R and from the fact that the transitive closure of R is irreflexive, it follows that a process must exist that is suspended on a communication with a process that is not suspended on a communication, and, since one of its guards is *true*, not suspended at all, which is a contradiction.

It follows that quiescence occurs only when all guards in the **if** statements of all processes are *false*. We first prove $\neg sent_{ij}$. If $sent_{ji}$ is *true* and all processes are quiescent a message has been sent on to_{ji} that has not been followed by the receipt of a message on $from_{ij}$. Had it been followed by such a message, the resulting communication between process i and j must have been completed since all processes are suspended, thus $sent_{ji}$ had been set to *false* again. The specification of the nonblocking channel guarantees us that a communication on *to* will result in $\overline{f}rom$ becoming *true*, therefore the situation in which $sent_{ij}$ is *true* precludes the processes from staying quiescent. $\forall(i,j : iRj : \neg c(v_i, v_{ij}) \ \wedge \ v_j = ov_{ji})$ and the invariant $\forall(i,j : iRj : v_{ij} = ov_{ji})$ implies $\forall(i,j : iRj : \neg c(v_i, v_j))$ which suffices to show that the old postcondition is established.

Proof of (2):

The four tuple $(bf, N(i,j : iRj : v_{ij} \neq v_j), N(i,j : jRi : sent_{ij} = (v_i = ov_{ij}), N(i,j : jRi : \mathbf{c}to_{ij}! - \mathbf{c}from_{ji}?))$ decreases for the forth alternative and for each other matching pair of alternatives in the program and is bounded from below. Checking this is not difficult, and it is left to the reader.

One last remark: using the nonblocking channel as a primitive, one can translate the algorithm in Figure 3. 3 directly. We chose to refine the algorithm in Figure 3.6, because it gives a solution with fewer channels.

3.6 A Load-Balancing Algorithm

In this section we show how a generalized version of the load-balancing algorithm of Hofstee, Lukkien, and van de Snepscheut [7] can be derived with this approach. A sequential program that establishes the required balance is given in Figure 3.9.

$\{\forall(i) :: p_i = P_i\}$
do $[\!]\ (i,j :: p_i < T \ \wedge \ p_j > T \rightarrow \ p_i, p_j := p_i + 1, p_j - 1)$ **od**
$\{\forall(i :: p_i \geq T) \vee \forall(i :: p_i \leq T)\}$

FIGURE 3.9.

In this program p_i represents the number of tasks in node i, and T a threshold. This program does not have the desired form, because iRj inserted in the range would be true for all pairs (i,j) and thus not represent a directed acyclic graph. We therefore choose a subset for our relation. If the subset is chosen in such a way that iRj represents a **rooted** directed acyclic graph that spans the original graph, we can rewrite the algorithm as in Figure 3.10.

$\{\forall(i) :: p_i = P_i\}$
do $[\!]\ (i,j : iRj : p_i < T \ \wedge \ p_j \geq T \rightarrow \ p_i, p_j := p_i + 1, p_j - 1)$
$[\!]\ \ [\!]\ (i,j : iRj : p_i > T \ \wedge \ p_j \leq T \rightarrow \ p_i, p_j := p_i - 1, p_j + 1)$
od
$\{\forall(i :: p_i \geq T) \vee \forall(i :: p_i \leq T)\}$

FIGURE 3.10.

Upon termination, a predecessor of a node with less than T tasks will have less than T tasks. By induction all predecessors of a node will have less than T tasks if that node has less than T tasks. Since the root is a predecessor of all other nodes in the graph, it will have less than T tasks upon termination if any node has less than T tasks upon termination. Similarly, the root will have more than T tasks if any node has more than T tasks. Since the two conditions are mutually exclusive, the original postcondition is established when all guards are false. The bound function for this program is $\sum_i w_i \ |p_i - T|$ where w_i is the longest distance to the

root. The function is bound by 0 and decreases on every iteration, hence the algorithm terminates.

We now have two alternatives, whereas the desired program only has one. Therefore we rewrite the algorithm once more to get the desired form. The inner **do**-statement may be replaced by an **if**-statement, but that is likely to be less efficient.

$\{\forall(i) :: p_i = P_i\}$
do $[\!]\ (i,j : iRj : (p_i < T \ \wedge \ p_j \geq T) \vee (p_i > T \ \wedge \ p_j \leq T) \rightarrow$
do $p_i < T \ \wedge \ p_j \geq T \rightarrow \ p_i, p_j := p_i + 1, p_j - 1$
$[\!]\ \ p_i > T \ \wedge \ p_j \leq T \rightarrow \ p_i, p_j := p_i - 1, p_j + 1$
od
od
$\{\forall(i :: p_i \geq T) \vee \forall(i :: p_i \leq T)\}$

FIGURE 3.11.

Now that the algorithm has the right form, we can apply the transformation to obtain the distributed solution in Figure 3.12.

The fifth alternative that we have added represents the consumption or generation of tasks by another process. Of course we can now no longer prove that the load will be balanced eventually, but it follows from our proofs that the load will become balanced across the network if the fifth alternative is no longer executed, and that actions that do not involve the fifth alternative lead to progress towards a balanced state. A complete solution should communicate tasks whenever the p_i are changed, the extension is trivial The solution derived here is more general than the solution presented in [7] since it allows the communication graph to contain cycles (but no directed cycles). The network can now be chosen in such a way that the root is not a bottleneck, even for very inhomogeneous problems. The solution can be generalized to more thresholds in exactly the same manner as the solution presented in [7].

3.7 Distributed Sorting

A program that sorts bags according to the relation R is given in Figure 3.13.

We see that it has the shape we required in Figure 3.1, and therefore the algorithm in Figure 3.14 is a correct refinement.

$$
\begin{array}{l}
\|(i :: \\
\quad ; (j : iRj : p_{ij} := 0) \ ; \\
\quad ; (h : hRi : op_{ih} := 0 \ ; \ \ sent_{ih} := false) \ ; \\
\quad \textbf{do } true \rightarrow \\
\qquad \textbf{if } [\!] \ (h : hRi : \overline{down_{ih}} \ \rightarrow \ sent_{ih} := false \ ; \\
\qquad\qquad\qquad\qquad\qquad\qquad (up_{ih}!p_i \| \ down_{ih}?x_i) \ ; \\
\qquad\qquad\qquad\qquad\qquad\qquad S_0''(x_i, p_i) \ ; \ \ op_{ih} := p_i) \\
\qquad [\!] \ [\!] \ (j : iRj : c(p_i, p_{ij}) \ \rightarrow \ (up_{ij}?x_i \| \ down_{ij}!p_i) \ ; \\
\qquad\qquad\qquad\qquad\qquad\qquad S_0'(p_i, x_i) \ ; \ \ p_{ij} := x_i) \\
\qquad [\!] \ [\!] \ (j : iRj : \overline{from_{ij}} \rightarrow \qquad from_{ij}?; \ \ (up_{ij}?x_i \| \ down_{ij}!p_i) \ ; \\
\qquad\qquad\qquad\qquad\qquad\qquad S_0'(p_i, x_i) \ ; \ \ p_{ij} := x_i) \\
\qquad [\!] \ [\!] \ (h : hRi : p_i \neq op_{ih} \ \wedge \ \neg sent_{ih} \rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad to_{ih}!; \ \ sent_{ih} := true \ ) \\
\qquad [\!] \ \overline{P}_i \rightarrow P_i?p_i \\
\qquad \textbf{fi} \\
\quad \textbf{od} \ ) \\
\| \ \ \|(i, j : jRi : nbchan(to'_{ij}, from_{ij}))
\end{array}
$$

$$
\begin{array}{l}
c(p_i, p_{ij}) \ = \ (p_i < T \ \wedge \ p_{ij} \geq T) \vee (p_i > T \ \wedge \ p_{ij} \leq T) \\
S_0'(p_i, p_j) \ = \ S_0''(p_i, p_j) \ = \ S_0(p_i, p_j) \ = \\
\textbf{if } c(p_i, p_j) \rightarrow \textbf{do } p_i < T \ \wedge \ p_j \geq T \rightarrow \ p_i, p_j := p_i + 1, p_j - 1 \\
\qquad\qquad\qquad\qquad [\!] \ \ p_i > T \ \wedge \ p_j \leq T \rightarrow \ p_i, p_j := p_i - 1, p_j + 1 \\
\qquad\qquad\qquad\qquad \textbf{od} \\
[\!] \ \neg c(p_i, p_j) \rightarrow \textbf{skip} \\
\textbf{fi} \\
= \\
\textbf{do } p_i < T \ \wedge \ p_j \geq T \rightarrow \ p_i, p_j := p_i + 1, p_j - 1 \\
[\!] \ \ p_i > T \ \wedge \ p_j \leq T \rightarrow \ p_i, p_j := p_i - 1, p_j + 1 \\
\textbf{od}
\end{array}
$$

FIGURE 3.12.

$\{\forall(i) :: b_i = B_i\}$

do $[\!]$ $(i, j : iRj : max(b_i) > min(b_j) \rightarrow$

$\quad b_i, b_j := b_i - max(b_i) + min(b_j), b_j - min(b_j) + max(b_i))$

od

FIGURE 3.13.

In this program we have made a slightly different choice for S_0' and S_0'' than in the load balancing example to avoid updating both bags in both nodes. It is easily verified that this choice satisfies the requirement on S_0' and S_0'' in Figure 3.5 in section 3.3. Inspection of the algorithm in Figure 3.14 reveals that even though whole bags are communicated, only their minimum or only their maximum is relevant. Thus it suffices to send a minimum or a maximum only. We modify the program accordingly and arrive at the solution in Figure 3.15.

Obviously the program can be improved by introducing variables to maintain the minimum and maximum of a bag. Also, we can leave out one of the communications in S_0' and S_0'', because the value communicated is never used. The program can even be modified to avoid both communications in the statements S_0' and S_0''. These communications can be left out if the statements S_0' and S_0'' and the following assignments to b_{ij} and ob_{ij} are reversed in order. The resulting program is then a refinement of the program in Figure 3.4 where the order of the statements S_0 and the multiple assignment directly following it has been reversed. The program is then still a refinement of the program in Figure 3.1. It has the same bound function and maintains the invariant of the corresponding refinements. However, the program will go through more iterations before terminating or becoming quiescent because the second part of the bound function increases by 1 more, as compared to the algorithm given, whenever the first part decreases.

As a final transformation we add variables that allow us to detect termination. This transformation is specific for the sorting problem. We assume that the bags are numbered from 0 to N and that the pairs $(i, i+1)$ for $0 \leq i < N$ form a subset of R, that is, we assume we want a total sort. Obviously, a termination condition for a process must be a stable condition, which presses us to look for monotonic variables. The reasoning is similar to the reasoning in [6]. The minimum and maximum of any individual bag may change nonmonotonically. However, we can prove that for any union of bags $lb_i = +(k : 0 \leq k \leq i \leq N : b_k)$ the maximum decreases monotonically, and for any union of bags $rb_i = +(k : 0 \leq i \leq k \leq N : b_k)$ the

$\{\forall(i :: b_i = B_i)\}$
$||(i ::$
; $(j : iRj : b_{ij} := \emptyset)$;
; $(h : hRi : ob_{ih} := \emptyset;\ sent_{ih} := false)$;
do $true \rightarrow$
if $[]\ (h : hRi : \overline{down_{ih}} \rightarrow$ $sent_{ih} := false$;
$(up_{ih}!b_i || down_{ih}?x_i)$;
$S_0''(x_i, b_i)$; $ob_{ih} := b_i)$
$[]\ []\ (j : iRj : c(b_i, b_{ij}) \rightarrow$ $(up_{ij}?x_i || down_{ij}!b_i)$;
$S_0'(b_i, x_i)$; $b_{ij} := x_i)$
$[]\ []\ (j : iRj : \overline{from_{ij}} \rightarrow$ $from_{ij}?;\ (up_{ij}?x_i || down_{ij}!b_i)$;
$S_0'(b_i, x_i)$; $b_{ij} := x_i)$
$[]\ []\ (h : hRi : b_i \neq ob_{ih} \wedge \neg sent_{ih} \rightarrow$
$to_{ih}!;\ sent_{ih} := true\ )$
fi
od
)
||
$||(i, j : jRi : nbchan(to'_{ij}, from_{ij}))$

$c(b_i, b_{ij}) = max(b_i) > min(b_{ij})$
$S_0'(b_i, x_i) =$ **if** $c(b_i, x_i) \rightarrow\ b_i := b_i - max(b_i) + min(x_i)$
$[]\neg c(b_i, x_i) \rightarrow$ **skip**
fi; $(up_{ij}?x_i || down_{ij}!b_i)$
$S_0''(x_i, b_i) =$ **if** $c(x_i, b_i) \rightarrow\ b_i := b_i - min(b_i) + max(x_i)$
$[]\neg c(x_i, b_i) \rightarrow$ **skip**
fi; $(up_{ih}!b_i || down_{ih}?x_i)$

FIGURE 3.14.

$\{\forall(i :: b_i = B_i)\}$
$\|(i ::$
$\quad ; (j : iRj : b_{ij} := \infty) ;$
$\quad ; (h : hRi : ob_{ih} := \infty; sent_{ih} := false) ;$
$\quad$ **do** $true \rightarrow$
$\quad\quad$ **if** $[\!]\ (h : hRi : \overline{down_{ih}} \rightarrow sent_{ih} := false ;$
$\quad\quad\quad\quad (up_{ih}!min(b_i) \| down_{ih}?x_i) ;$
$\quad\quad\quad\quad S_0''(x_i, b_i) ;\ ob_{ih} := min(b_i))$
$\quad\quad [\!]\ [\!]\ (j : iRj : c(b_i, b_{ij}) \rightarrow (up_{ij}?x_i \| down_{ij}!max(b_i)) ;$
$\quad\quad\quad\quad S_0'(b_i, x_i) ;\ b_{ij} := x_i)$
$\quad\quad [\!]\ [\!]\ (j : iRj : \overline{from_{ij}} \rightarrow from_{ij}?;\ (up_{ij}?x_i \| down_{ij}!max(b_i)) ;$
$\quad\quad\quad\quad S_0'(b_i, x_i) ;\ b_{ij} := x_i)$
$\quad\quad [\!]\ [\!]\ (h : hRi : min(b_i) \neq ob_{ih} \ \wedge\ \neg sent_{ih} \rightarrow$
$\quad\quad\quad\quad to_{ih}!;\ sent_{ih} := true\ )$
$\quad\quad$ **fi**
$\quad$ **od**
$)$
$\|$
$\|(i, j : jRi : nbchan(to'_{ij}, from_{ij}))$

$c(b_i, b_{ij}) = max(b_i) > b_{ij}$
$S_0'(b_i, x_i) =$ **if** $max(b_i) > x_i \rightarrow b_i := b_i - max(b_i) + x_i$
$\quad [\!]\neg max(b_i) \leq x_i \rightarrow$ **skip**
$\quad$ **fi**; $(up_{ij}?x_i \| down_{ij}!max(b_i))$
$S_0''(x_i, b_i) =$ **if** $x_i > min(b_i) \rightarrow b_i := b_i - min(b_i) + x_i$
$\quad [\!]\ \ x_i \leq min(b_i) \rightarrow$ **skip**
$\quad$ **fi**; $(up_{ih}!min(b_i) \| down_{ih}?x_i)$

FIGURE 3.15.

minimum increases monotonically. We can easily verify this by inspecting Figure 3.13. The minima and maxima change by exchanging elements between bags only. If both bags are in one of the sets described above, or both are not, its minimum, or maximum, does not change. If one bag is in the set, and the other is not, then if $i < j$ only (i, j) can be in R since adding (j, i) to R would create a directed cycle. This implies that in that case the minimum of bag j only increases and the maximum of bag i only decreases.

We introduce variables that approximate the maxima and minima of the sets introduced above. The following invariant is maintained: $\forall(i, j : iRj : LM_{ji} = PM_{ij} \geq max(lb_i)) \ \wedge \ \forall(i, j : jRi : rm_{ji} = pm_{ij} \leq min(rb_i))$. We arrive at the solution in Figure 3.16.

Verifying the equalities in the invariant is easy, they hold initially, and every assignment to PM or pm is followed by a pair of communications that restores the invariant. The other two relations in the invariant hold because none of the PM_{ij} or pm_{ij} can be assigned a value that does not satisfy the relation, unless one of the other variables PM_{ij} or pm_{ij} did not satisfy the relation. This follows from the form of the assignments to the pm and PM. Since all pm and PM are guaranteed to satisfy the relations in the invariant initially, the invariant is maintained.

Given this invariant, and the old invariant, we prove that the final program satisfies the original specification:

(1) Termination of all processes implies the old postcondition.

(2) Not all processes are suspended. (Unless they have all terminated.)

(3) Each matching pair of alternatives decreases a bound function.

Proof of (1):

$$
\begin{aligned}
& \forall(i, j : jRi : LM_{ij} \leq pm_{ij}) \ \wedge \ \forall(i, j : iRj : PM_{ij} \leq rm_{ij}) \\
\Rightarrow & \{\text{Invariant} : PM_{ij} = LM_{ji}\} \\
& \forall(i, j : iRj : LM_{ji} \leq rm_{ij}) \\
\equiv & \{\text{Invariant} : LM_{ji} \geq max(lb_i) \ \wedge \ rm_{ij} \leq min(rb_j)\} \\
& \forall(i, j : iRj : max(lb_i) \leq LM_{ji} \leq rm_{ij} \leq min(rb_j)) \\
\Rightarrow & \{max(b_i) \leq max(lb_i), min(rb_j) \leq min(b_j)\} \\
& \forall(i, j : iRj : max(b_i) \leq min(b_j))
\end{aligned}
$$

Proof of (2); Not all processes are suspended or terminated, unless they have all terminated:

First we claim that a process i that has not terminated and is blocked on a communication has a successor j such that iRj holds and j has the same property. Since R is acyclic and finite, this implies that no process is blocked on a communication. The reasoning is virtually identical to the

$\{\forall(i :: b_i = B_i)\}$

$||(i ::$

$\quad ; (j : iRj : b_{ij}, rm_{ij}, PM_{ij} := +\infty, -\infty, +\infty) ;$

$\quad ; (h : hRi : ob_{ih}, sent_{ih}, LM_{ih}, pm_{ih} := +\infty, false, +\infty, -\infty) ;$

$\quad$ **do** $\exists(j : jRi : LM_{ij} > pm_{ij}) \vee \exists(j : iRj : PM_{ij} > rm_{ij}) \rightarrow$

$\quad\quad$ **if** $[\!]\ (h : hRi : \overline{down_{ih}} \rightarrow sent_{ih} := false\ ;$

$\quad\quad\quad\quad pm_{ih} := min(b_i + +(k : iRk : rm_{ik}));$

$\quad\quad\quad\quad (up_{ih}!(min(b_i), pm_{ih})||\ down_{ih}?(x_i, LM_{ih}))\ ;$

$\quad\quad\quad\quad S_0''(x_i, b_i)\ ;\ ob_{ih} := min(b_i))$

$\quad\quad [\!]\ [\!]\ (j : iRj : max(b_i) > b_{ij} \wedge PM_{ij} > rm_{ij} \rightarrow$

$\quad\quad\quad\quad PM_{ij} := max(b_i + +(k : kRi : LM_{ik}));$

$\quad\quad\quad\quad (up_{ij}?(x_i, rm_{ij})||\ down_{ij}!(max(b_i), PM_{ij}))\ ;$

$\quad\quad\quad\quad S_0'(b_i, x_i)\ ;\ b_{ij} := x_i)$

$\quad\quad [\!]\ [\!]\ (j : iRj : \overline{from_{ij}} \wedge PM_{ij} > rm_{ij} \rightarrow from_{ij}?;$

$\quad\quad\quad\quad PM_{ij} := max(b_i + +(k : kRi : LM_{ik}));$

$\quad\quad\quad\quad (up_{ij}?(x_i, rm_{ij})||\ down_{ij}!(max(b_i), PM_{ij}))\ ;$

$\quad\quad\quad\quad S_0'(b_i, x_i)\ ;\ b_{ij} := x_i)$

$\quad\quad [\!]\ [\!]\ (h : hRi : min(b_i) \neq ob_{ih} \wedge \neg sent_{ih} \rightarrow$

$\quad\quad\quad\quad to_{ih}!;\ sent_{ih} := true\ )$

$\quad\quad [\!]\ [\!]\ (j : iRj : max(b_i + +(k : kRi : LM_{ik})) \neq PM_{ij} \wedge PM_{ij} > rm_{ij} \rightarrow$

$\quad\quad\quad\quad PM_{ij} := max(b_i + +(k : kRi : LM_{ik}));$

$\quad\quad\quad\quad (up_{ij}?(x_i, rm_{ij})||\ down_{ij}!(max(b_i), PM_{ij}))\ ;$

$\quad\quad\quad\quad S_0'(b_i, x_i)\ ;\ b_{ij} := x_i)$

$\quad\quad [\!]\ [\!]\ (h : hRi : min(b_i + +(k : iRk : rm_{ik})) \neq pm_{ih} \wedge \neg sent_{ih} \rightarrow$

$\quad\quad\quad\quad to_{ih}!;\ sent_{ih} := true\ )$

$\quad\quad$ **fi**

$\quad$ **od**

$)$

$||\ \ ||(i, j : jRi : nbchan(to'_{ij}, from_{ij}))$

S_0' and S_0'' as before

FIGURE 3.16.

reasoning in section 3.5, with one important exception: because processes may now terminate, we also have to show that no process is committed to a communication with a process that has terminated. The conjuncts $PM_{ij} > rm_{ij}$ with the invariant $PM_{ij} = LM_{ji} \ \wedge \ rm_{ij} = pm_{ji}$ in the second, third, and fifth alternative guarantee just that. The rest of the proof is left to the reader.

Next we show that if not all bags are sorted, not all processes are suspended or have terminated:

$$
\begin{aligned}
& \exists(i,j : iRj : max(b_i) > min(b_j)) \\
\Rightarrow\ & \{\text{Invariants}\} \\
& \exists(i,j : iRj : PM_{ij} \geq max(lb_i) \geq max(b_i) > min(b_j) \geq min(rb_j) \geq pm_{ji}) \\
\Rightarrow\ & \{\text{Invariants}\} \\
& \exists(i,j : iRj : PM_{ij} > rm_{ij} \ \wedge \ LM_{ji} > pm_{ji} \\
& \ \wedge \ (max(b_i) > b_{ij} \vee ob_{ji} \neq min(b_j))) \\
\Rightarrow\ & \{\text{guards}\} \\
& \text{process } i \text{ and process } j \text{ not terminated and} \\
& (\text{process } i \text{ not suspended or process } j \text{ not suspended or } sent_{ji} = true)
\end{aligned}
$$

We had concluded before that the situation in which $sent_{ji}$ is true and process i is not terminated eventually leads to process i not being suspended.

The following step is to show that if all bags are sorted and all processes are suspended on the alternative statement or have terminated, the property $max(b_i) = max(b_i + \ +(k : kRi : LM_{ik}))$ holds for all processes that have not terminated. We prove it holds for all processes by induction on the graph without directed cycles that R represents. The induction step requires us to prove that a node for which all predecessors satisfy the property, satisfies the property also:

$$\forall(j : jRi : LM_{ij} \leq pm_{ij} \vee$$
$$(LM_{ij} > pm_{ij} \ \wedge \ max(b_j) = max(b_j + +(k : kRj : LM_{jk}))))$$
$$\Rightarrow \{\text{Invariants}\}$$
$$\forall(j : jRi : LM_{ij} \leq pm_{ij} \vee$$
$$(PM_{ji} > rm_{ji} \ \wedge \ max(b_j) = max(b_j + +(k : kRj : LM_{jk}))))$$
$$\Rightarrow \{ \text{ process } j \text{ is suspended, guard of fifth alternative } \}$$
$$\forall(j : jRi : LM_{ij} \leq pm_{ij} \vee$$
$$(PM_{ji} > rm_{ji} \ \wedge \ max(b_j) = max(b_j + +(k : kRj : LM_{jk})) = PM_{ji}))$$
$$\Rightarrow \{\text{Invariants, sortedness}\}$$
$$\forall(j : jRi : LM_{ij} \leq pm_{ij} \leq min(b_i) \vee LM_{ij} = max(b_j) \leq min(b_i))$$
$$\Rightarrow$$
$$max(b_i) = max(b_i + \ +(j : jRi : LM_{ij}))$$

Finally we show that if all bags are sorted, and if all processes that have not terminated are suspended and have the above property, then any process i that has not terminated has a successor j such that iRj holds, and j has not terminated. The fact that R is finite and acyclic then implies that no process is suspended, thus all processes have terminated, which is what we set out to prove.

$$
\begin{aligned}
&\exists(j : iRj : PM_{ij} > rm_{ij}) \vee \exists(j : jRi : LM_{ij} > pm_{ij}) \\
\Rightarrow &\{ \text{ inv., all proc. suspended } (sent = false, \text{ guard of 6th alternative } \} \\
&\exists(j : iRj : PM_{ij} > rm_{ij}) \vee \\
&\exists(j : jRi : PM_{ji} > rm_{ji} \ \wedge \ pm_{ij} = min(b_i + \ +(k : iRk : rm_{ik})) \\
\Rightarrow &\{ \text{ invariant, all proc. suspended, guard of fifth alternative } \} \\
&\exists(j : iRj : PM_{ij} > rm_{ij}) \vee \\
&\exists(j : jRi : PM_{ji} > rm_{ji} \ \wedge \ pm_{ij} = min(b_i + \ +(k : iRk : rm_{ik})) \ \wedge \\
&\qquad PM_{ji} = max(b_j + \ +(k : kRi : LM_{ik}))) \\
\Rightarrow &\{ \text{ invariant, } max(b_i) = max(b_i + \ +(k : kRi : LM_{ik})), \text{ sortedness} \\
&\exists(j : iRj : PM_{ij} > rm_{ij}) \vee \\
&\exists(j : jRi : LM_{ij} = max(b_j) > pm_{ij} = min(b_i + \ +(k : iRk : rm_{ik})) \\
&\qquad \wedge \ max(b_j) \leq min(b_i)) \\
\Rightarrow & \\
&\exists(j : iRj : PM_{ij} > rm_{ij}) \vee \exists(k : iRk : min(b_i) > rm_{ik}) \\
\Rightarrow &\{PM_{ik} \geq max(lb_i) \geq max(b_i) \geq min(b_i)\} \\
&\exists(j : iRj : PM_{ij} > rm_{ij})
\end{aligned}
$$

Proof of (3); Each matching pair of alternatives decreases a bound function:

The four tuple

$$
\begin{aligned}
&(old - bf, \\
&N(i,j : iRj : v_{ij} \neq v_j) + \ +(i,j : iRj : PM_{ij}) - \ +(i,j : jRi : pm_{ji}), \\
&N(i,j : jRi : \neg sent_{ij} \ \wedge \ \neg(min(b_i) = ob_{ij})) + \\
&\quad N(i,j : jRi : \neg sent_{ij} \ \wedge \ \neg(min(b_i + \ +(k : iRk : rm_{ik})) = pm_{ij})), \\
&N(i,j : jRi : \mathbf{c}to_{ij}! - \mathbf{c}from_{ji}?))
\end{aligned}
$$

decreases for the fourth and fifth alternative in the program and for each of the three matching pairs of alternatives. For the sake of this variant function we have to read $+\infty$ and $-\infty$ in the program as 'some finite number bigger than anything else' and 'some finite number smaller than everything else' respectively.

3.8 Discussion

This paper was written in an attempt to unify the derivation of a distributed algorithm for load balancing [7], with the derivation of a new series of distributed sorting algorithms [8] that the paper and a previous paper on distributed sorting [6] inspired. Essential to the derivation is realizing that since we do not put a limit on the number of processes that can interact with any process and change its variables, information in any process about any other process is bound to be volatile. This problem can be overcome by imposing a high degree of synchronization, but, unless one is willing to make further assumptions about the specified problem, such a solution may be inefficient. Assuming that information is volatile, has the further advantage that introducing other processes that modify variables in one or more of the processes, as in the load balancing algorithm [7], does not present any difficulties. For the proof that quiescence is ultimately achieved in that case, one must make the assumption that these other processes are quiescent at some point in time. Other than that the proofs remain the same.

Since the load balancing algorithm does not terminate, we chose to treat termination as a late refinement step. The nonterminating version of the sorting algorithm is not without merit either, it can most likely be used, for instance, in a distributed database application to keep a changing collection of bags sorted 'on the fly'. Since the specification in Figure 3.1 is quite general, it seems likely that several other problems can be cast in this form as well.

3.9 Related Work

The sequential program that specifies the problem we try to distribute, is similar to a class of programs known as 'Action Systems' [10] or 'Unity Programs' [9].

The motivation for and goals of the first two refinements steps, those that maintain a sequential program, differs from [10] only in that we require the guards to involve variables local to one process only. This is related to the fact that we do not allow output guards in our distributed programs. In general, output guards cannot be implemented efficiently. More stringent conditions on the guards and structure of the sequential programs can ensure that a translation into communicating sequential processes can be generated automatically [11]. This method has the advantage of staying in the domain of sequential programs throughout the whole derivation. It is unclear, however, to what extent the restrictions on the final sequential program affect the efficiency of the final distributed algorithm or the ability of the programmer to find an efficient solution.

Quiescence of our distributed algorithms is related to the fixpoint of a

'Unity Program'. Methods for fixpoint detection described in [9] will therefore be helpful when trying to detect quiescence and thus create terminating algorithms.

Acknowledgements: I thank my advisor, Jan van de Snepscheut, for encouraging me to do something more methodical than another example of a derivation; I hope I have succeeded. Jan, Klaas Esselink, and Rustan Leino did most of the formalization of the nonblocking channel. Johan Lukkien invented the implementation of the nonblocking channels on the spot when, in a group meeting at Caltech, I expressed my doubts that it could be done without sharing variables between processes. I also thank Jan, Johan and Alain Martin, for work on papers that inspired the present one. I thank Ralph-Johan Back for lectures and discussions on program refinement. Audiences at T. U. Eindhoven, Koninklijke/Shell-Laboratorium Amsterdam, and Groningen University had several questions and comments that have led to improvements in the paper. Finally, I thank Jan, Rustan, Ulla Binau, and Wim Feijen for reading the manuscript and many helpful comments.

References

[1] E. W. Dijkstra, *A discipline of programming* (Prentice-Hall, Englewood Cliffs, NJ, 1976).

[2] C.A.R. Hoare, Communicating sequential processes, *Comm. ACM* (1978) 666-677.

[3] C.A.R. Hoare, *Communicating sequential processes* (Prentice-Hall International Series in Computer Science, 1985).

[4] A.J. Martin, An axiomatic definition of synchronization primitives, *Acta Informatica* **16** (1981) 219-235.

[5] A.J. Martin, The probe: an addition to communication primitives, *Information Processing Letters*, **20** (1985) 125-130.

[6] H. P. Hofstee, A. J. Martin and J. L. A. van de Snepscheut, Distributed sorting, *Science of Computer Programming* **15** (1990) 119-133.

[7] H. P. Hofstee, J. J. Lukkien and J. L. A. van de Snepscheut, A distributed implementation of a taskpool, In *Research Directions in High-Level Parallel Programming Languages*, J.B. Banâtre and D. Le Métayer (Eds.), LNCS 574 (1992) 338-348.

[8] H. P. Hofstee, Distributed sorting revisited, *unpublished.*

[9] K. Mani Chandy and J. Misra, *Parallel program design, a foundation* (Addison-Wesley, Reading, MA, 1988).

[10] R.-J.R. Back and R. Kurki-Suonio, Decentralization of process nets with centralized control, *Distributed Computing*, (1989) 3:73-87.

[11] R.-J.R. Back and K. Sere, Deriving an occam implementation of action systems. In *Proc. of the 3rd Refinement Workshop BCS FACS*, J.M. Morris and R. C. Shaw (Eds.), (Springer Verlag Workshops in Computing Series, 1991).

4

Some experimental results.

H. Peter Hofstee [1]

ABSTRACT In this note I present an analysis of the complexity of the algorithms in [0,1] and experimental results of both the distributed sorting and taskpool [4,5] algorithms.

4.1 Distributed sorting

I will refer to the algorithm in figure 1.4 of as **dsort** (distributed sort), to the algorithm in figure 1.5 of [0] as **edsort** (efficient distributed sort), and to the algorithm in Figure 3[2] of [1] as **2dsort** (2-dimensional distributed sort). The total number of elements to be sorted is M, the number of processors is N, and the number of elements per processor is k ($M = k \cdot N$).

I will assume that bag operations can be done in $log(k)$ time. This can be achieved at a cost proportional to $k \cdot log(k)$ for initializing an AVL-tree implementation of the bags.

All algorithms have space requirements proportional to the number of elements is each bag, therefore we analyze the time complexity only. I give a worst case, in terms of sortedness of the given bags, analysis only.

The minimum time complexity of all algorithms is the number of swaps times the cost of each swap divided by the number of swaps that can go in parallel. For the three algorithms this amounts to

- **dsort:**

$$k \cdot N \cdot N \cdot (\mathrm{A}_d \cdot log(k) + \mathrm{B}_d) + \mathrm{C} \cdot k \cdot log(k)$$

$$\equiv$$

$$M \cdot N \cdot (\mathrm{A}_d \cdot log(M/N) + \mathrm{B}_d) + \mathrm{C} \cdot M/N \cdot log(M/N)$$

- **edsort:**

$$k \cdot N \cdot (\mathrm{A}_e \cdot log(k) + \mathrm{B}_e) + \mathrm{C} \cdot k \cdot log(k)$$

$$\equiv$$

$$M \cdot (\mathrm{A}_e \cdot log(M/N) + \mathrm{B}_e) + \mathrm{C} \cdot M/N \cdot log(M/N)$$

---

[1] H.Peter Hofstee is supported by an IBM graduate fellowship.

[2] This corresponds to the algorithm given in figure 3.16 with the communications in S_0' and S_0'' removed as described in section 3.7 and $R(i,j)$ chosen to represent a 2-dimensional network.

- **2dsort:**

$$k \cdot \sqrt{N} \cdot (\mathrm{A}_2 \cdot log(k) + \mathrm{B}_2) + \mathrm{C} \cdot k \cdot log(k)$$

$$\equiv$$

$$M/\sqrt{N} \cdot (\mathrm{A}_2 \cdot log(M/N) + \mathrm{B}_2) + \mathrm{C} \cdot M/N \cdot log(M/N)$$

The cost of initializing the data structure to store the bags is independent of the sorting algorithm, but the amount of communication, and the extent to which communication and computation may interleave is algorithm dependent. Therefore the constants A and B differ per algorithm, whereas C is algorithm independent. In a system where the communication cost dominates (true in our experiments), the second of all three pairs of formulae shows that for a given amount of data, using more processors is counterproductive for the first algorithm, of marginal help with the second (speedup is at most a factor $log(N)$), and sensible with the third algorithm (speedup is a factor of $\sqrt{(}N)$).

Since the algorithms we have developed are smooth, they are most useful in situations where the bags have to be resorted after incremental updates, or in a dynamic situation.

In the formulae above we have not taken into account the time needed to load and store the data or the program. If the computer is large enough, and connected to a single source for the data, this term, which is proportional to M will in fact dominate. On the other hand, one can envision several applications where the data are created distributed over all the processors, and may be processed further in the processor where they end up after the sort.

Because the range of values I used was limited (0 through 1000), the factor $log(k)$ in the first term did not show up in the experimental results. In the original version the bags were implemented as an unsorted array, so that this was a factor of k rather than $log(k)$, and in that case it did have a significant effect. Since the last term is the same for all three algorithms we measured it separately, and we started the clock for the other measurements after the data had been stored in an AVL-tree. The constant C is in fact a slowly decreasing function of k. For our experiments on the S2010 it ranged from 0.045 ms (k=16) to 0.031 ms (k=512).

The following table shows some experimental results on the (192 node) S2010 using message passing C [2] and the experimental MOSAIC system at Caltech using MOSAIC pascal[3], as well as a comparison with the theoretical functions. The data match the expected behavior as a function of N very well.

The main discrepancy between the data and the expected values is that the last column, which should be constant, in fact increases with N indicating that the complexity estimate was somewhat optimistic. On the other hand, the ratio seems to approach a constant asymptotically, and the range of values is small (a factor of 2 versus a factor of 70 in N).

S2010

times in ms k=256

#procs	dsort	t_d/N^2	edsort	t_e/N	2dsort	$t_2/\sqrt{N}$
2	165	41	266	133		
4	713	45	912	228	972	486
8	2944	46	2071	259		
9	3792	47	2329	259	1926	642
16	11803	46	4187	262	3293	823
25			6562	262	4480	896
36			9339	259	5355	893
49			12532	256	6479	926
64			16332	255	7386	923
81			20542	254	8904	989
100			25127	251	9723	972
121			30338	251	10914	992
144			35900	249	11736	978

MOSAIC 28 nodes

times in ms k=256

#procs	dsort	t_d/N^2	edsort	t_e/N	2dsort	$t_2/\sqrt{N}$
2	40	10	60	30		
4	100	6	140	35	120	60
8	250	4	260	33		
9			300	33	260	87
16	690	3	500	31	400	100
25			760	30	520	104

Finally I compare our algorithms to two well-known solutions.

- **Odd-Even transposition sort.**
  The complexity of odd-even transposition sort [Manber p.390] is the same as that of **edsort**. The main advantage is that the processors will always be synchronized, the main disadvantage is that the expected complexity is the same as the worst case complexity.

- **Mergesort.**
  If a double tree structure of depth $log(M)$ is used (one tree to split the data, one tree to merge them), a speedup of $log(N)$ can be achieved. The main advantage of mergesort is its throughput, the main disadvantages are that storage proportional to the size of M rather than

N is needed in at least some processors, and that the algorithm is not smooth.

4.2 Taskpool

We have implemented the taskpool [4,5] in Mosaic Pascal [2] and used it to do the load balancing for a functional language that is evaluated concurrently [6]. We have looked at three different problems: a matrix multiplication example, computing generalized fibonacci numbers, and an algorithm that computes the solution to the missionaries and cannibals problem [7]. Two different versions of the taskpool, one where the connecting graph is a tree cf.[4], and one where the connecting graph is a directed mesh cf.[5], have been compared to a placement scheme where each task that is generated is sent to a randomly chosen processor. When the load is small, we expect a relatively large overhead for the taskpool algorithms, the overhead should be proportional to the depth of the rooted directed acyclic graph. For heavier loads, that is when the number of elements stays between thresholds, we expect less overhead from the taskpool than from the random placement approach, which communicates all tasks between processors. It should be noted that random placement distributes tasks more or less evenly, but does not necessarily distribute the work evenly; a process that is computing a large task is as likely to get another as any other process. The simple random placement algorithm that we have used here cannot easily be extended to guarantee that bufferspace in the whole network runs out before the computation halts, whereas this can be accomplished by the taskpool algorithm by choosing another threshold one less than the buffer size.

Random placement has another, perhaps more fundamental problem. If no information about other processors is maintained, it is not possible to put a bound on the number of messages sent to any particular processor at any particular time. This implies that an unbounded amount of bufferspace must be allocated since the routing network (algorithm) used is only deadlock free if any message arriving at a node is taken from the network. In particular no guarantee is given that a send will succeed unless all incoming messages are taken from the network. Therefore schemes that deflect messages from nodes that are filling up are not deadlock free if bufferspace is limited.

The measurements are in general agreement with the observations above. If the workload is small, as is the case in the matrix multiply and missionaries examples, the random placement algorithm performs better for larger number of processors. If the workload is high, as is the case in the generalized fibonacci example, the taskpool algorithms maintain their advantage over random placement. Furthermore, the random placement algorithm runs out of bufferspace in some of the larger examples.

MOSAIC matrix(n*n) multipy

times in ms

#procs	n	random placement	taskpool tree	taskpool mesh
1	4	1840	1370	1380
2	4	890	750	720
4	4	460	510	490
8	4	300	470	410
15	4	250	470	400
1	8	10390	7770	7770
2	8	5330	4020	3950
4	8	2670	2440	2260
8	8	1460	1890	1620
15	8	1040	1690	1220
1	16			
2	16	33390	24210	23870
4	16	16230	13910	12930
8	16	8280	9360	7810
15	16	5470	7640	6350

MOSAIC fib4(n)

times in ms

#procs	n	random placement	taskpool tree	taskpool mesh
1	10	540	360	360
2	10	290	180	180
4	10	140	110	110
8	10	80	70	70
15	10	50	60	60
1	14	7520	5010	5020
2	14	4240	2520	2530
4	14	1940	1280	1280
8	14	940	700	670
15	14	620	470	390
1	19	200320	132370	132590
2	19	out(2)	67140	67120
4	19	out(1)	33800	33750
8	19	out(6)	17200	17060
15	19	out(8)	10000	9530

MOSAIC cannibals & missionaries
times in ms

#procs	random placement	taskpool tree	taskpool mesh
1	19430	12620	12880
2	8470	6940	6660
4	4290	5110	4300
8	2880	4920	3570
15	2480	4970	2990

Acknowledgements: Thanks are due to Charles L. Seitz for letting me play with his machines, and to Wen-King Su for making sure I could use them.

References

[0] H. P. Hofstee, A. J. Martin, J. L. A. van de Snepscheut, Distributed Sorting, *Science of Computer Programming* 15 (1990) 119-133

[1] H. P. Hofstee, Distributed Sorting Revisited, *Unpublished*

[2] C. L. Seitz, J. Seizovic, Wen-King Su, The C Programmers Abbreviated Guide to Multicomputer Programming, *Caltech Internal Report*, CS-TR-88-1

[3] J. J. Lukkien, J.L.A. van de Snepscheut, A Tutorial Introduction to Mosaic Pascal, *Caltech Internal Report*, CS-TR-91-2

[4] H. P. Hofstee, J. J. Lukkien and J. L. A. van de Snepscheut, A distributed implementation of a taskpool, In *Research Directions in High-Level Parallel Programming Languages*, J.B. Banâtre and D. Le Métayer (Eds.), LNCS 574 (1992) 338-348.

[5] H. P. Hofstee, Distributing a class of sequential programs, Ch.3, this report, to be published in Mathematics of Program Construction 1992, conference proceedings.

[6] J.L.A. van de Snepscheut, private communication.

[7] F.E.J. Kruseman Aretz, On deriving a LISP program from its specification, *Science of Computer Programming* 10 (1988) 19-31