Science of Computer Programming 15 (1990) 159-169
North-Holland

TWO SMALL PARALLEL PROGRAMMING EXERCISES

L.D.J.C. LOYENS and J.G.G. VAN DE VORST
Koninklijke/Shell-Laboratorium, Amsterdam (Shell Research B.V.), P.O. Box 3003, 1003 AA Amsterdam, Netherlands

Revised June 1990

Abstract. The aim of the present article is to show how parallel programs can be derived from functional specifications. Each program consists of the parallel composition of a number of instances of a single parameterised process. The formulation of parameterised invariants is the central issue in the derivations.

Introduction

Two small parallel programs are derived from functional specifications in the style of the series Small Programming Exercises of this journal. The first exercise is a parallel program computing all partial sums of a given array. The second exercise is a typical example of a class of "segment" problems, a parallel program computing the maximum length of any right-maximal segment.

As the use of invariants has proved to be fruitful in sequential programming [2, 3], and nothing seems to prohibit the extension of their use to parallel programming [1, 8], we will use invariants to derive our programs.

Here, a parallel program consists of p, $0<p$, instances of a single parameterised process. Each instance is identified by a number q, $0 \leq q<p$. Instance q of the parameterised process (or process q for short) establishes an instance of the local postcondition which itself is also parameterised by q. The conjunction of these postconditions implies the postcondition of the parallel program. From the local postcondition a parameterised local invariant is derived, much as in sequential programming.

In general, a process needs to communicate with other processes. Which values are communicated by a process can be deduced easily from the local invariants.

Other examples of this approach can be found in [5, 6, 9]. The parallel programs we end up with are not systolic (like in [1]), because the programs consist of p instances of a single parameterised process. Furthermore, the parallelism arises from concurrent operations on distributed data. Before presenting the two exercises we will introduce the notation used in this paper.

The notation is borrowed from [7], and is based on Dijkstra's guarded command language [2]. We have extended the notation with primitives to express parallel

composition and communication. The construct

par $q : 0 \leq q < p : S(q)$ **rap**

denotes the parallel composition of p instances of the parameterised process S. If the range of process number q is omitted, then it is $0 \leq q < p$.

Communication between processes is explicit and is established by sending and receiving messages via point-to-point channels like in CSP [4]. The statement $q!e$ denotes the sending of the value of expression e to process q. Receiving a value x from process q is denoted by $q?x$. In contrast to CSP, we only require that the messages sent along a channel arrive in the sending order and that no messages get lost.

We do not use shared variables in our programs. Each program variable is local to a single process. This is not reflected in the variable names but it will be clear from the context which process is meant.

1. First exercise: Parallel partial sums

A simple parallel program that computes the partial sums of an array f of length n is given. The arrays involved are distributed across the p processes. We choose the following distribution: array element $f(i)$, $0 \leq i < n$, is assigned to process i **mod** p.

The introduction of this "cyclic" distribution as part of the problem prohibits, in our view, a clear operational picture of a parallel solution. We will demonstrate that the derivation is straightforward and that it relies entirely on the use of parameterised invariants. The choice of the communication network will also be part of the derivation.

We start with a functional specification of the parallel composition of p instances of a parameterised process.

$$
\begin{array}{l}
[\![\, p, n : int; \{0 < p \ \wedge \ n \bmod p = 0\} \\
\quad f(i : 0 \leq i < n) : \textbf{array of } int; \\
\quad [\![h(j : 0 \leq j < n) : \textbf{array of } int; \\
\qquad \textbf{par } q : 0 \leq q < p : S(q) \textbf{ rap} \\
\qquad \{R : (\mathrm{A}j : 0 \leq j < n : h(j) = (\Sigma i : 0 \leq i \leq j : f(i)))\} \\
\quad]\!] \\
]\!]
\end{array}
$$

The restriction n **mod** $p = 0$ in the functional specification is introduced for the sake of simplicity. A similar derivation can be given if n **mod** $p \neq 0$. The parameterised process to be designed is $S(q)$. Every process will manipulate the elements of arrays f and h that are local to it, and will establish a local postcondition. The conjunction of all local postconditions will imply the postcondition R of the program.

All array elements of f and h with indices i, i **mod** $p = q$, are local to process q. This cyclic distribution and the postcondition R suggest the following local postcondition $R(q)$ for process q.

$$R(q): \quad (\mathbf{A}j : 0 \leq j < n \ \wedge \ j \ \mathbf{mod}\ p = q : h(j) = sum(j)) \quad \text{where}$$
$$sum(j) = (\Sigma i : 0 \leq i \leq j : f(i)) \text{ for all } 0 \leq j < n$$

We rewrite the local postcondition to simplify its range.

$$R(q): \quad (\mathbf{A}j : 0 \leq j < m : h(j * p + q) = sum(j * p + q)) \quad \text{where}$$
$$m = n \ \mathbf{div}\ p$$

This formulation can be used to find an invariant $P(q)$.

$$P(q): \quad 0 \leq k \leq m \ \wedge \ (\mathbf{A}j : 0 \leq j < k : h(j * p + q) = sum(j * p + q))$$

The invariant $P(q)$ is found in the normal way by replacing the constant m in the postcondition $R(q)$ by a variable k. Every process will initialise its k by setting it to zero, and progress will be made by incrementing k by one. The resulting process body is a loop with $k \neq m$ as guard. Consider $P(q)(k := k + 1)$, i.e., $P(q)$ with k replaced by $k + 1$:

$$\begin{aligned}
& P(q)(k := k+1) \\
\equiv\ & 0 \leq k+1 \leq m \ \wedge \ (\mathbf{A}j : 0 \leq j < k+1 : h(j * p + q) = sum(j * p + q)) \\
\Leftarrow\ & \{\text{split of } j = k, \text{ calculus, definition } P(q)\} \\
& k \neq m \ \wedge \ P(q) \ \wedge \ h(k * p + q) = sum(k * p + q)
\end{aligned}$$

From this little calculation it follows that the value of $sum(k * p + q)$ has to be computed. We now rewrite $sum(k * p + q)$ as a sum of p terms, taking into account the distribution of f, and isolating the expressions local to process q.

$$\begin{aligned}
& sum(k * p + q) \\
=\ & \{\text{definition } sum\} \\
& (\Sigma i : 0 \leq i \leq k * p + q : f(i)) \\
=\ & \{\text{split of } i = k * p + q, \text{ calculus}\} \\
& (\Sigma x : 0 \leq x < p \\
& \quad : (\Sigma i : 0 \leq i < k * p + q \ \wedge \ i \ \mathbf{mod}\ p = x : f(i))) + \\
& f(k * p + q) \\
=\ & \{\text{definition } psum \text{ (see below)}\} \\
& (\Sigma x : 0 \leq x < p : psum(k * p + q, x)) + f(k * p + q) \\
=\ & \{\text{range splitting } x < q,\ x = q,\ q < x\} \\
& (\Sigma x : 0 \leq x < q : psum(k * p + q, x)) + psum(k * p + q, q) + \\
& (\Sigma x : q < x < p : psum(k * p + q, x)) + f(k * p + q) \\
=\ & \{\text{definition } prev, next \text{ (see below)}\} \\
& prev(k * p + q, q) + psum(k * p + q, q) + \\
& f(k * p + q) + next(k * p + q, q)
\end{aligned}$$

Where, for $0 \leq b < n$,

$$psum(b, q) = (\Sigma i : 0 \leq i < b \ \wedge \ i \ \mathbf{mod} \ p = q : f(i))$$
$$prev(b, q) = (\Sigma x : 0 \leq x < q : psum(b, x))$$
$$next(b, q) = (\Sigma x : q < x < p : psum(b, x))$$

In this way, $sum(k * p + q)$ is written as a local sum (referring only to array elements local to process q) and two global sums *prev* and *next* (referring to array elements that are not available in process q). The local sum $psum(k * p + q, q) + f(k * p + q)$ can be computed easily. Two arrays $h0$ and $h1$ are introduced in which the accumulated partial sums of the first q processes and the accumulated partial sums of the last $p - q - 1$ processes are stored.

$R0(q)$: $(\mathbf{A}j : 0 \leq j < m : h0(j * p + q) = prev(j * p + q, q))$
$R1(q)$: $(\mathbf{A}j : 0 \leq j < m : h1(j * p + q) = next(j * p + q, q))$

The structure of $S(q)$ is

$$\begin{array}{l}
[\![s, k : int;\ h0, h1(j : 0 \leq j < n) : \mathbf{array\ of}\ int; \\
\quad S0(q); S1(q)\ \{R0(q) \ \wedge \ R1(q)\} \\
\quad ; s, k := 0, 0 \\
\quad ; \mathbf{do}\ k \neq m \rightarrow \\
\qquad \{s = psum(k * p + q, q) \ \wedge \ P(q)\} \\
\qquad h(k * p + q) := h0(k * p + q) + s + f(k * p + q) + h1(k * p + q) \\
\qquad ; s, k := s + f(k * p + q), k + 1 \\
\quad \mathbf{od} \\
]\!]
\end{array}$$

In the program we have separated the local computation part from the global computation part $(S0(q); S1(q))$. The global computation will consist of one or more communication phases. In this way, a clear distinction in the derivation is achieved. We will now focus on the construction of a program that establishes $R0(q)$. An invariant is derived just as for $R(q)$:

$$\begin{array}{ll}
P0(q): & 0 \leq k \leq m \ \wedge \ (\mathbf{A}j : 0 \leq j < k : h0(j * p + q) = prev(j * p + q, q)) \\
& P0(q)(k := k + 1) \\
\equiv & 0 \leq k + 1 \leq m \ \wedge \\
& (\mathbf{A}j : 0 \leq j < k + 1 : h0(j * p + q) = prev(j * p + q, q)) \\
\Leftarrow & \{\text{split of } j = k, \text{ calculus, definition } P0\} \\
& k \neq m \ \wedge \ P0(q) \ \wedge \ h0(k * p + q) = prev(k * p + q, q)
\end{array}$$

The value of $prev(k * p + q, q)$ has to be computed. From the definition of *prev* the following property is obtained:

$$\begin{array}{ll}
& prev(k * p + q, q) \\
= & \{0 \leq k < m\} \\
& prev((k - 1) * p + q, q) + (\Sigma x : 0 \leq x < q : f(k * p + x))
\end{array}$$

Hint: use the definitions for *prev* and *psum*, the empty sum convention and some elementary calculus. Note that $prev((-1) * p + q, q) = 0$.

Summing q array elements of the first q processes is needed in order to restore $P0(q)$. This can be done using a chain communication nework and designing a communication process that uses this chain. In a chain communication network, process q can communicate with processes $q+1$ and $q-1$, if they exist. The resulting program for $S0(q)$ is

```
|[r, t, k: int;
   r, k := 0, 0
 ; do k ≠ m →
      {r = prev((k − 1) * p + q, q) ∧ P0(q)}
      if q = 0 → t := 0 □ q > 0 → (q − 1)?t fi
      {t = (Σx: 0 ≤ x < q: f(k * p + x))}
      {t + f(k * p + q) = (Σx: 0 ≤ x < q + 1: f(k * p + x))}
    ; if q < p − 1 → (q + 1)! t + f(k * p + q)
      □ q = p − 1 → skip
      fi
    ; h0(k * p + q) := r + t
    ; r, k := r + t, k + 1
   od
]|
```

The communication process establishing $R1(q)$ is similar to $S0(q)$, and is not given. Note that the processes $S0(q)$ and $S1(q)$ can be executed in parallel. We have chosen to record all values of *prev* and *next* in the arrays $h0$ and $h1$, respectively. This simplifies the derivation by having a global computation part and a local computation part. So much for the derivation of parameterised process $S0(q)$.

The communications in $S0(q)$ and $S1(q)$ are done in two pipes of length p of opposite directions (from process 0 to $p-1$ and vice versa). Hence, the communication network used is a chain. The number of messages communicated in each pipe is $m = n$ **div** p, all communications are sent one after another, and it takes $O(p)$ communications to reach the end of a pipe. The time complexity of $S0(q)$ and $S1(q)$ is $O(n/p+p)$ and the memory usage is $O(n/p)$. The overall complexity of the parallel program is equal to the complexity of a process instance $S(q)$, which is $O(n/p+p)$. For $p=1$, S reduces to a sequential program, and time complexity is $O(\sqrt{n})$ for $p=\sqrt{n}$. Thus, under the condition $p \leq \sqrt{n}$, S shows a linear speedup.

The derivation of the parallel program was carried out on the assumption of a cyclic distribution. Explicit use of the properties of this distribution was made, yielding a parallel program consisting of a communication phase and a computation phase. If we would have made use of a distribution that assigns consecutive segments to processes, another parallel program would have been obtained whose time complexity is the same.

In the derivation, it was essential to separate the local expressions that can be computed within a process from the global terms. This "isolation" technique is the

lesson to be learned from this exercise. In the following exercise we will apply this technique again.

2. Second exercise: A parallel segment problem

This exercise is inspired by [7, Exercise 27]. It is a typical example of a segment problem. Given is an array f of length n. The problem is to compute the maximum length of any right-maximal segment. The distribution of the array f across the p processes is not specified a priori. The specification of the communication network is also left open.

```
|[p, n : int; {0 < p ≤ n}
 f(i : 0 ≤ i < n) : array of int;
 par q : 0 ≤ q < p :
     |[r : int;
       S(q)
       {q ≠ 0 ∨
        r = (max i, j : 0 ≤ i ≤ j < n ∧ less(i, j, f(j)) : j − i + 1)}
     ]|
 rap
]|
```

For $0 \leq i \leq j \leq n$ and any integer y:

$$less(i, j, y) \equiv (\mathbf{A}k : i \leq k < j : f(k) \leq y)$$

In the postcondition of $S(q)$ it is stated that process $q = 0$ has a local variable r whose value equals the maximal length of a right-maximal segment, i.e. a segment for which the predicate *less* holds. In contrast to the previous exercise nothing is said about the distribution of f across the p processes. This means that we cannot directly formulate a local postcondition in terms of variables local to a process. Instead, we will first generalise the expression for r and then derive a suitable distribution for f.

For $0 \leq a \leq b \leq n$,

$$rm(a, b) = (\mathbf{max}\ i, j : a \leq i \leq j < b \ \wedge \ less(i, j, f(j)) : j - i + 1)$$

We will split up the computation of rm for two consecutive segments $[a, b)$ and $[b, c)$ into computations for each of these segments and one computation for the continuation segment specified below. The expressions for $[a, b)$ and $[b, c)$ are local, in the sense that they only refer to either $[a, b)$ or $[b, c)$. For the continuation segment, the expression is global, but it will appear feasible to divide it into local subexpressions. The expression $rm(a, c)$ for a non-empty segment $[a, c)$ of f can be rewritten as follows.

Let $0 \leq a < b < c \leq n$, then

$$rm(a, c) = rm(a, b) \textbf{ max } (tl(a, b, m(b, c)) + hd(b, c)) \textbf{ max } rm(b, c) \quad (0)$$

where

$$m(b, c) = (\textbf{max } i : b \leq i < c : f(i))$$
$$hd(b, c) = (\textbf{max } i : b \leq i < c \wedge m(b, c) = f(i) : i - b + 1)$$
$$tl(a, b, y) = (\textbf{max } i : a \leq i < b \wedge less(i, b, y) : b - i)$$

Here, we take the maximum of the empty set for *rm*, *hd* and *tl* to be zero.

We will give an informal explanation of the above formulas, a proof of (0) is given later. Consider two consecutive non-empty segments. The expression for $rm(a, c)$ is split up into expressions for the segments $[a, b)$ and $[b, c)$, and in one expression for the continuation segment. The continuation segment consists of a tail segment (ending in $f(b-1)$) and a head segment (starting in $f(b)$), is itself right-maximal, and has maximal length. The head segment is also right-maximal. Clearly, the maximum to the right of the head segment must be the maximum of the segment $[b, c)$. The longest right-maximal head segment is given by $hd(b, c)$ and it ends in element $f(hd(b, c) + b - 1) = m(b, c)$. This longest right-maximal head segment can be extended further to the left if the elements in the tail segment are smaller than or equal to $m(b, c)$. The longest right-maximal tail segment relative to $m(b, c)$ is given by $tl(a, b, m(b, c))$. Hence, the length of the longest right-maximal continuation segment is $tl(a, b, m(b, c)) + hd(b, c)$.

It is straightforward to see that the following formulas also hold:

$$m(a, c) = m(a, b) \textbf{ max } m(b, c)$$
$$hd(a, c) = hd(a, b) \quad \text{if } m(a, c) \neq m(b, c) \quad (1)$$
$$hd(a, c) = b - a + hd(b, c) \quad \text{if } m(a, c) = m(b, c)$$

A short proof of the formula for *rm* is given.

Proof. The following properties are used in the proof:

$$less(a, c, y) \equiv less(a, b, y) \wedge less(b, c, y) \quad \text{for any } y \quad (2)$$
$$f(hd(b, c) + b - 1) = m(b, c) \quad (3)$$
$$less(b, hd(b, c) + b - 1), m(b, c)) \text{ holds} \quad (4)$$
$$less(b, b, m(b, c)) \text{ holds} \quad (5)$$
$$(\textbf{max } j : b \leq j < c \wedge less(b, j, m(b, c)) : j - b + 1) = hd(b, c) \quad (6)$$

$rm(a, c)$
$=$ {definition *rm*}
$(\textbf{max } i, j : a \leq i \leq j < c \wedge less(i, j, f(j)) : j - i + 1)$
$=$ {range splitting, definition *rm*}
$rm(a, b) \textbf{ max }$
$(\textbf{max } i, j : a \leq i \leq b \wedge b \leq j < c \wedge less(i, j, f(j)) : j - i + 1) \textbf{ max }$
$rm(b, c)$

On the one hand,

$$
\begin{aligned}
& (\mathbf{max}\ i,j : a \leq i \leq b \ \wedge\ b \leq j < c \ \wedge\ less(i,j,f(j)) \\
& \qquad : j-i+1) \\
= \ & \{\text{property (2)}\} \\
& (\mathbf{max}\ i,j : a \leq i \leq b \ \wedge\ less(i,b,f(j)) \ \wedge \\
& \qquad b \leq j < c \ \wedge\ less(b,j,f(j)) \\
& \qquad : j-i+1) \\
\geq \ & \{\text{take } j = hd(b,c)+b-1, \text{ hence } b \leq j < c, \text{ properties (3), (4)}\} \\
& (\mathbf{max}\ i : a \leq i \leq b \ \wedge\ less(i,b,m(b,c)) : hd(b,c)+b-i) \\
= \ & \{\text{calculus split of } i=b, \text{ property (5), definition } tl\} \\
& tl(a,b,m(b,c))+hd(b,c)
\end{aligned}
$$

On the other hand,

$$
\begin{aligned}
& (\mathbf{max}\ i,j : a \leq i \leq b \ \wedge\ b \leq j < c \ \wedge\ less(i,j,f(j)) : j-i+1) \\
\leq \ & \{(\mathbf{A}j : b \leq j < c : f(j) \leq m(b,c))\} \\
& (\mathbf{max}\ i,j : a \leq i \leq b \ \wedge\ b \leq j < c \ \wedge\ less(i,j,m(b,c)) \\
& \qquad : j-i+1) \\
= \ & \{\text{property (2), calculus}\} \\
& (\mathbf{max}\ i,j : a \leq i \leq b \ \wedge\ b \leq j < c \ \wedge \\
& \qquad less(i,b,m(b,c)) \ \wedge\ less(b,j,m(b,c)) \\
& \qquad : (b-i)+(j-b+1)) \\
= \ & \{\text{calculus}\} \\
& (\mathbf{max}\ i : a \leq i \leq b \ \wedge\ less(i,b,m(b,c)) : b-i) + \\
& (\mathbf{max}\ j : b \leq j < c \ \wedge\ less(b,j,m(b,c)) : j-b+1) \\
= \ & \{\text{split of } i=b, \text{ property (5), definition } tl, \text{ property (6)}\} \\
& tl(a,b,m(b,c))+hd(b,c) \qquad \square
\end{aligned}
$$

Two things follow: a suitable distribution of the array f across the p processes and the program text. The decomposition of rm suggests that each process q be given a segment of the array f. For the distribution of the array f we use a function $l, l : \{0..p\} \to \{0..n\}$, l is strictly increasing, $l(0)=0$, and $l(p)=n$. Process q has segment $f(i : l(q) \leq i < l(q+1))$ as a local variable. To avoid excessive indexing with l in the expressions for rm, hd, tl and m, the process numbers are used as indices, so $rm(q, q+1)$ stands for $rm(l(q), l(q+1))$.

We reformulate the expression for $rm(a,c)$ in terms of local and global expressions for every process q by taking $a=l(q)$, $b=l(q+1)$ and $c=l(p)$ in (0):

$$
\begin{aligned}
rm(q,p) = \ & rm(q,q+1)\ \mathbf{max} \\
& (tl(q,q+1,m(q+1,p))+hd(q+1,p))\ \mathbf{max} \\
& rm(q+1,p)
\end{aligned}
$$

First, every process computes its local rm, hd and m (referring to $[l(q), l(q+1)]$).

With these values, the global expressions $hd(q+1, p)$ and $m(q+1, p)$ can be evaluated. Then each process uses $m(q+1, p)$ to compute $tl(q, q+1, m(q+1, p))$. Finally, by repeated application of the above formula, the global expressions $rm(q, p)$ can be evaluated.

We will give the program text and the complexity. In the complexity estimates, it is assumed that n **mod** $p = 0$ and that $l(q) = q * (n/p)$, $0 \leq q \leq p$.

The structure of $S(q)$ is

$$
\begin{array}{l}
[\![\, s, t, sn, tn, u : int; \\
\quad S0(q) \\
\quad \{r = rm(q, q+1) \ \wedge \ s = hd(q, q+1) \ \wedge \ t = m(q, q+1)\} \\
\; ; S1(q) \\
\quad \{sn = hd(q+1, p) \ \wedge \ tn = m(q+1, p)\} \\
\; ; S2(q) \\
\quad \{u = tl(q, q+1, tn)\} \\
\; ; S3(q) \\
\quad \{r = rm(q, p)\} \\
]\!]
\end{array}
$$

The process invariant of $S0(q)$ is

$$P0(q): \quad l(q) \leq k \leq l(q+1) \ \wedge \ r = rm(k, l(q+1)) \ \wedge \ s = hd(k, l(q+1))$$

$S0(q)$ has complexity $O(l(q+1) - l(q)) = O(n/p)$. The program text of $S0(q)$ is

$$
\begin{array}{l}
[\![k : int; \\
\quad k, r, s := l(q+1), 0, 0 \, \{l(q) < l(q+1) \ \wedge \ P0(q)\} \\
\; ; \textbf{do } k \neq l(q) \rightarrow \\
\qquad k := k - 1 \\
\qquad ; \textbf{if } f(k) \leq f(k+s) \rightarrow s := s + 1 \\
\qquad \quad \square f(k) > f(k+s) \rightarrow s := 1 \\
\qquad \quad \textbf{fi} \\
\qquad ; r := r \ \textbf{max} \ s\{P0(q)\} \\
\quad \textbf{od} \ \{s \geq 1\} \\
\; ; t := f(l(q) + s - 1) \\
\quad \{r = rm(q, q+1) \ \wedge \ s = hd(q, q+1) \ \wedge \ t = m(q, q+1)\} \\
]\!]
\end{array}
$$

Using the formulas (1), process q, $q < p-1$, can compute $hd(q, p)$ and $m(q, p)$, provided that $hd(q+1, p)$ and $m(q+1, p)$ are given. This results in a chain communication network. Process $q+1$, $q < p-1$, communicates $hd(q+1, p)$ and $m(q+1, p)$ to process q. For $q = p-1$, $hd(q+1, p) = hd(l(p), l(p)) = 0$, and $m(q+1, p) =$

$m(l(p), l(p)) = -\infty$. The complexity of $S1(q)$ is $O(p)$. The program text of $S1(q)$ is:

```
|[ h0, h1 : int;
   if q = p − 1 → sn := 0; tn := −∞
   □ q < p − 1 → (q + 1)? sn, tn
   fi
   {sn = hd(q + 1, p) ∧ tn = m(q + 1, p)}
  ; if t ≤ tn → {m(q, p) = m(q + 1, p)}
      h0 := l(q + 1) − l(q) + sn; h1 := tn
   □ t > tn → {m(q, p) ≠ m(q + 1, p)}
      h0 := s; h1 := t
   fi
   {h0 = hd(q, p) ∧ h1 = m(q, p)}
  ; if q > 0 → (q − 1)!h0, h1
   □ q = 0 → skip
   fi
]|
```

$S2(q)$ is again a sequential process. The process invariant of $S2(1)$ is

$P2(q)$: $l(q) \leq k \leq l(q+1) \wedge u = tl(l(q), k, tn)$

$S2(q)$ has complexity $O(n/p)$. The program text of $S2(q)$ is

```
|[ k : int;
   k, u := l(q), 0 {P2(q)}
  ; do k ≠ l(q + 1) →
      if f(k) ≤ tn → u := u + 1 □ f(k) > tn → u := 0 fi
     ; k := k + 1 {P2(q)}
   od
]|
```

Finally, process q, $q < p-1$, can compute $rm(q, p)$, provided that $rm(q+1, p)$ is given. Again, this leads to a chain communication network. Process $q+1$, $q < p-1$, communicates $rm(q+1, p)$ to process q. For $q = p-1$, $rm(p, p) = 0$. The complexity of $S3(q)$ is $O(p)$.

```
|[ h : int;
   if q = p − 1 → h := 0 □ q < p − 1 → (q + 1)? h fi
   {h = rm(q + 1, p)}
  ; r := r max (u + sn) max h
   {r = rm(q, p)}
  ; if q > 0 → (q − 1)! r □ q = 0 → skip fi
]|
```

Adding the complexities of processes $S0(q)$, $S1(q)$, $S2(q)$ and $S3(q)$, we conclude that the complexity of S is $O(n/p+p)$. For $p=1$, S reduces to a sequential program, and for $p=\sqrt{n}$, S has complexity $O(\sqrt{n})$. Thus, under the condition $p \leq \sqrt{n}$, S shows a linear speedup.

3. Final remarks

In this paper the formal development of two small parallel programs has been presented. Both programs consist of the parallel composition of p instances of a parameterised process. The first program computes the partial sums of a given array f. Right at the start of the derivation, it is decided to distribute f in a cyclic way across the p process instances. This decision heavily influenced the derivation and the resulting program text. It has appeared to be difficult to postpone the choice of the distribution of f and to produce a program text that is independent of the distribution for this example program.

The second parallel program computes the maximum length of any right-maximal segment. The choice of the distribution of the array is made at a later point in the derivation and it follows naturally from the problem analysis. The distribution is parameterised with a distribution function l, which is also present in the final program text. It is possible to give similar derivations for segment problems like the Plateau Problem (see [3, p. 203]) and the maximal sum of any segment of an array.

Both programs are developed employing predicate calculus in the usual way. As in the sequential programming we use invariants, but we have parameterised the invariants with the process number q. The communication processes are trivial and their specifications follow directly from the problem analysis. In both programs we compute repeatedly an expression by taking the sum or maximum of p terms using a chain network. The complexities of the two parallel programs can be reduced further from $O(n/p+p)$ to $O(n/p+\log p)$ if we compute the sum or maximum of these terms using another communication network.

References

[1] K.M. Chandy and J. Misra, Systolic algorithms as programs, *Distributed Comput.* **1** (1986) 177-183.
[2] E.W.D. Dijkstra, *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, NJ, 1976).
[3] D. Gries, *The Science of Programming* (Springer, New York, 1981).
[4] C.A.R. Hoare, *Communicating Sequential Processes* (Prentice-Hall International, London, 1985).
[5] L.D.J.C. Loyens, Parallel programming techniques for linear algebra, in: *Parallel Computing 1988*, Lecture Notes in Computer Science **384** (Springer, New York, 1989) 32-43.
[6] L.D.J.C. Loyens and R.H. Bisseling, The formal construction of a parallel triangular system solver, in: J.L.A. van de Snepscheut, ed., *Mathematics of Program Construction*, Lecture Notes in Computer Science **375** (Springer, New York, 1989) 325-334.
[7] M. Rem, Small programming exercises 11, *Sci. Comput. Programming* **6** (1986) 313-318.
[8] J.L.A. van de Snepscheut, A derivation of a distributed implementation of Warshall's algorithm, *Sci. Comput. Programming* **7** (1986) 55-60.
[9] J.G.G. van de Vorst, The formal development of a parallel program performing LU-decomposition, *Acta Inform.* **26** (1988) 1-17.