Quaestiones Informaticae

An official publication of the Computer Society of South Africa
‘n Amptelike tydskrif van die Rekenaarvereniging van Suid-Afrika

Editor: Prof G. Wiechers
Department of Computer Science and Information Systems
University of South Africa
P.O. Box 392, Pretoria 0001

Editorial Advisory Board
PROFESSOR D. W. BARRON
Department of Mathematics
The University
Southampton S09 5NH
England

PROFESSOR K. GREGGOR
Computer Centre
University of Port Elizabeth
Port Elizabeth 6001
South Africa

PROFESSOR K. MACGREGOR
Department of Computer Science
University of Cape Town
Private Bag
Rondebosch 7700
South Africa

PROFESSOR M. H. WILLIAMS
Department of Computer Science
Herriot-Watt University
Edinburgh
Scotland

MR P. P. ROETS
NRIMS
CSIR
P.O. Box 395
Pretoria 0001
South Africa

PROFESSOR S. H. VON SOLMS
Department of Computer Science
Rand Afrikaans University
Auckland Park
Johannesburg 2001
South Africa

DR H. MESSERSCHMIDT
IBM South Africa
P.O. Box 1419
Johannesburg 2000

MR P. C. PIROW
Graduate School of Business Administration
University of the Witwatersrand
P.O. Box 31170
Braamfontein 2017
South Africa

Subscriptions
Annual subscriptions are as follows:

<table>
<thead>
<tr>
<th></th>
<th>SA</th>
<th>US</th>
<th>UK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Individuals</td>
<td>R6</td>
<td>$7</td>
<td>£3,0</td>
</tr>
<tr>
<td>Institutions</td>
<td>R12</td>
<td>$14</td>
<td>£6,0</td>
</tr>
</tbody>
</table>

Quaestiones Informaticae is prepared for publication by Thomson Publications South Africa (Pty) Ltd for the Computer Society of South Africa.
NOTE FROM THE EDITOR

After an absence of two years we are happy to announce that we are now in a position to continue the publication of Quaestiones Informaticae. The first Volume of QI consists of three numbers, and appeared during the period June 1979 till March 1980 under the editorship of Prof Howard Williams. Because Prof Williams took up a post at the Herriott-Watt University in Edinburgh, he had to relinquish his position as editor. The Computer Society of South Africa, which sponsors the publication of QI, appointed me as editor, whereas Mr Peter Pirow took over the administration of the Journal. The editorial board functions under the auspices of the Publications Committee of the CSSA.

The current issue is Number 1 of Volume 2. It is planned to publish altogether three issues in the Volume, with most of the papers coming from the Second South African Computer Symposium on Research in Theory, Software and Hardware. This Symposium was held on 28th and 29th October, 1981. At present it appears that most of the material published in this Journal comes from papers read at conferences. We invite possible contributors to submit their work to QI, since only the vigorous support of researchers in the field of Computer Science and Information Systems will keep this publication alive.

G WIECHERS

November, 1983
A CSP Description of some Parallel Sorting Algorithms

M. H. Linck
University of Cape Town, Rondebosch, Cape

Abstract
Hoare’s CSP notation is used to describe 3 parallel sorting algorithms. The first algorithm uses \( n/2 \) processes working in parallel, the second uses an array of \( n \) parallel processes and the third algorithm is a parallel version of quicksort.

Introduction
During the past 5 years several high level languages or notations have been developed that allow for parallelism in the description of algorithms.

The languages Concurrent Pascal of Brinch Hansen[1], Modula of Wirth[2] and Pascal Plus of Welsh and Burstard[3], are all based on Pascal[4]. All these languages allow for the description of processes running in parallel; however communication between processes has to be effected via the use of a special monitor[5] process.

CSP[6] developed by Hoare builds on Dijkstra’s[7] guarded command notation for both alternative and repetitive commands. In CSP a program is described as a number of processes running in parallel which, if they communicate with each other, do so by special Input-Output commands which are strictly synchronized.

Sorting is an area of prime importance in computing and Computer Science. This paper describes 3 sorting algorithms, written in CSP. All these algorithms involve parallelism in their description.

Before describing these new algorithms, the familiar BUBBLESORT[8] algorithm is described in CSP, see Figure 1, in order to introduce many of the concepts of this new language[9]. Bubblesort is described as 3 processes running in parallel:-

- SENDER — which sends the values to be sorted, one after another, to BUBBLESORT.
- BUBBLESORT — which accepts these values, sorts them, and then sends the sorted values to RECEIVER.
- RECEIVER — receives the sorted values.

Three sorting algorithms involving parallelism
In this section three parallel sorts are described. For each sort the algorithm is described, a CSP description is given as well as a small worked numerical example. Some comments are made about each method.

```csp
(Sender ::

comment 100 values are sent, one after another, from SENDER to BUBBLESORT;

i := 1;
\*i ≤ 100 ⇒ BUBBLESORT! value (i) ; i := i + 1

//BUBBLESORT ::
i, size, limit, no-of-swops ; integer;
temp : real;
list : (1 . . . 100) real;
i := 1; \*SENDER ? list(i) ⇒ i := i + 1;
size := i ; limit := i ; no-of-swops := 1;
\*limit ≥ 2 ; no-of-swops ≠ 0 ⇒
i := 1 ; no-of-swops := 0;
\*i ≤ limit -1 ⇒
[ list(i) ≤ list(i + 1) ⇒ skip
  list(i) > list(i + 1) ⇒ temp := list(i);
  list(i) := list(i + 1);
  list(i + 1) := temp;
  no-of-swops := no-of-swops + 1
];
i := i + 1;
] lim := limit -1;

i := 1; \*i ≤ size ⇒ RECEIVER!list(i); i := i + 1

//RECEIVER ::
j : integer; sorted : (1 . . . 100) real;
j := 1; \*BUBBLESORT ? sorted(j) ⇒ j := j + 1; .........
```

7
2. A Sorting algorithm using n/2 processes in parallel

Algorithm: This algorithm consists of 2 cycles described below as CYCLE1 and CYCLE2 respectively. These cycles are performed, one after another, as many times as necessary. The algorithm terminates when two successive cycles, (CYCLE1 and CYCLE2) or (CYCLE2 or CYCLE1), both give rise to no swaps.

CYCLE1: The 1st & 2nd value is compared by process1 & swapped if necessary.
The 3rd & 4th value is compared by process2 & swapped if necessary.
The 5th & 6th value is compared by process3 & swapped if necessary.

CYCLE2: The 2nd & 3rd value is compared by process1 & swapped if necessary.
The 4th & 5th value is compared by process2 & swapped if necessary.

etc....
The n-2th & n-1th value is compared by process (n/2-1) & swapped if necessary.

ALL THESE STEPS ARE PERFORMED IN PARALLEL.

(If n is even then the last value is just omitted from the comparison for this cycle).

A CSP description of this algorithm, for n even, is given in Figure 2 and a small worked numerical example is given in Figure 3.

Each cycle of this sort will require n/2 processors, (i.e. one for each process running in parallel). Each process performs a comparison and a swap (if necessary). In the average case, where the values to be sorted are randomly distributed, \( \frac{1}{2}(n + 1) \) cycles of operation will be required to successfully complete the sort. In the worst case, with the values initially being in anti-sorted order, \( (n + 1) \) cycles of operation will be required.

The time to perform this sort will therefore be \( O(n) \).

Figure 2. A CSP description of a n/2 parallel processor sort (n = 50)

Unsorted Sorted

<table>
<thead>
<tr>
<th>19</th>
<th>7</th>
<th>7</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>19</td>
<td>2</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>19</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>5</td>
<td>19</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>19</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>19</td>
<td>19</td>
<td>19</td>
</tr>
</tbody>
</table>

cycle1 cycle2 cycle1 cycle2 cycle1 cycle2 cycle1
No-of swops 1 2 2 2 2 0 0

Figure 3. A worked numerical example of a n/2 (n = 6) parallel processor sort.
3. A sorting algorithm using an array of \( n \) parallel processes.

This algorithm has an array of \( n \) parallel processes, in essence one for each value being sorted. There are two extra processes, one that injects the unsorted values into the 1st process of the array of processes and one that collects the sorted values from the \( n' \)th process.

Each process \( i \) of the array of \( n \) processes is described as follows: Each process is sent exactly \( n \) values by process \( i-1 \).

1. It keeps the 1st value sent to it. Let us call this the comparitor value.

For each subsequent value sent to the process

\[
\text{SORTER}(0) :: i : \text{integer};
\text{unsorted} : (1 \ldots 100) \text{real};
\begin{align*}
\text{comment} & \text{this process sends the 100 unsorted values to SORTER}(1); \\
i & := 1; \\
\text{LET} 1 \leq 100 \rightarrow \text{SORTER}(1)\text{unsorted}(i) ; i := i + 1
\end{align*}
\]

\[
\text{/\text{SORTER}} (i : 1 \ldots 100) ::
value1,value2 : \text{real};
\text{SORTER}(i-1)? value1;
\begin{align*}
\text{LET} & \text{SORTER}(i-1)? value2 \rightarrow \begin{cases}
\text{value2} \geq \text{value1} & \text{SORTER}(i + 1)? \text{value2} \\
\text{value2} < \text{value1} & \text{SORTER}(i + 1)? \text{value1} \\
\text{value1} := \text{value2}
\end{cases}
\end{align*}
\]

\[
\text{SORTER}(i + 1)? \text{value1}
\]

\[
\text{/\text{SORTER}} (101) ::
i : \text{integer};
\text{sorted} : (1 \ldots 100) \text{real};
i := 1; \\
\text{LET} \text{SORTER}(100)? \text{sorted}(i); i := i + 1
\]

Figure 4. A CSP description for a sort, for 100 values, using an array of 100 parallel processes.

2. It compares this value with the comparitor value.
3. It sends the larger of the two values to process \( i+1 \).
4. It sets the comparitor value equal to the smaller of the two values.

Finally, after it has passed on \( n-1 \) values to process \( i+1 \)
5. It sends the \( n' \)th value to process \( i+1 \), and
6. Terminates.

A CSP description of this algorithm is given in Figure 4 and a small worked numerical example is given, step by step, in Figure 5. The use of an array of \( n \) processes, to sort \( n \) values, leads to a particularly concise and elegant description of this method.

Unsorted Sorted

Figure 5. Numerical example illustrating, step by step, a Sort using an array of 3 parallel processes to sort 3 values into order.
Each value to be sorted passes, in turn, through \( n \) identical processes. Each process consists essentially of accepting a value, comparing two values, swapping two values (if necessary) and sending a value to the succeeding process. The time to perform this sort will therefore be \( O(n) \).

### 4. A Parallel version of Quicksort

Quicksort\(^{[8,10]} \) is a well known and very fast sorting algorithm. A comparator value, from the values being sorted, is chosen. The values being sorted are then partitioned into two groups: Group1 with values less than the comparator value and Group2 with values greater than or equal to the comparator value. The position of the comparator value, in the sorted array, is computed. Quicksort is then invoked recursively, first for Group1 then for Group2.

The parallel version of Quicksort essentially replaces the recursive calls of the sequential solution with the parallel execution of 'son' processes.

Again an array of \( n \) parallel processes is used.

Each of these \( n \) parallel processes may be described as follows:

1. It accepts one or more values from its father process.
2. The first value so obtained is kept as the comparator value, \( (CV) \), for that process.
3. Each subsequent value so obtained is compared with \( CV \) and sent either to the left-hand-son process or the right-hand-son process depending on whether it is less than or greater than or equal to \( CV \). (The connection between the father process and any son process is made immediately before the father process sends the first value to that son process).
4. A value, called count, is kept of the number of values sent to the left-hand-son process.
5. After a process has received all the values to be sorted it then receives, from its father process, a value:- old-count.
6. The position of this process's \( CV \), in the final sorted array, is now computed:
   \[
   \text{pos} = \text{old-count} + \text{count} + 1
   \]
7. This value of pos is sent to the right-hand-son process.
8. The value of \( CV \) together with the value of pos is sent to the collector process.
9. The value of old-count is sent to the left-hand-son process.
10. The process terminates.

A CSP description of this algorithm is given in Figure 6 and a small worked example is given in Figure 7. Figure 7a illustrates the values being sorted and Figure 7b illustrates the computation of the position of each process's comparator value in the final sorted array.

This method is an adaptation of Gries's solution\(^{[6]} \) to the Prime number problem using the Sieve of Eratosthenes\(^{[11]} \). Irrespective of the distribution of the values to be sorted all the values have to pass through the 1st process before continuing on. This takes \( n \) cycles of operation. The last value, can at most pass through a further \( n-1 \) processes (taking a further \( n-1 \) cycles of operation). The maximum time for the sort is thus \( n + (n-1) = 2n-1 \) cycles of operation. Each cycle (or process) essentially consists of: accepting a value from the previous process, a comparison and sending a value on to some successor process. The time to perform this sort will therefore by \( O(n) \).

### 5. Conclusion

Three parallel sorting algorithms have been presented. All these algorithms will work faster than comparable non-parallel sorts because more processes are used. The complexity of all these algorithms is \( O(n) \). Further comparison between these methods and between parallel and non-parallel methods is required. This is intended as a future development.

CSP is shown to be a powerful notation, capable of concise description of parallel algorithms.

CSP's concept of an array of processes, was directly responsible for the construction of the algorithm for method2. This is an excellent example of a good construct in a notation leading to new and novel solutions to a problem.

### Acknowledgements

I wish to thank Prof. Steve Schach for his constructive comments.

\[
\text{CSP is shown to be a powerful notation, capable of concise description of parallel algorithms.}
\]

\[
\text{CSP's concept of an array of processes, was directly responsible for the construction of the algorithm for method2.}
\]

\[
\text{This is an excellent example of a good construct in a notation leading to new and novel solutions to a problem.}
\]

(Please turn to next page)
father, left-son, right-son, count, old-count : integer;
value1, value2 : real;
left-son-created, right-son-created : boolean;
left-son-created := false; right-son-created := false;
PROCESS-NO? father; comment father process number is obtained;
S(father)? value1; comment the comparitor value is obtained;
count := 0;
\[ S(father)? value2 -> \]
\[ value1 > value2 -> \]
\[ left-son-created \Rightarrow S(left-son)! value2; \]
\[ count := count + 1 \]
\[ right-son-created \Rightarrow PROCESS-NO!(); \]
\[ PROCESS-NO? left-son; \]
\[ S(left-son)! value2; \]
\[ left-son-created := true; \]
\[ count := count + 1 \]
\[ \]
\[ \]
\[ left-son-created \Rightarrow S(left-son)! value2 \]
\[ count := count + 1 \]
\[ \]
\[ right-son-created \Rightarrow PROCESS-NO!(); \]
\[ PROCESS-NO? right-son; \]
\[ S(right-son)! value2; \]
\[ right-son-created := true; \]
\[ \]
S(father)? old-count;
count := old-count + count + 1;
comment count is position of value1 in the sorted array;
\[ right-son-created \Rightarrow S(right-son)! count \]
\[ right-son-created \Rightarrow skip \]
\[ \]
S(0)! (value1, count);
\[ left-son-created \Rightarrow S(left-son)! old-count \]
\[ left-son-created \Rightarrow skip \]
\[ \]

Figure 6. A CSP description of a parallel version.

Figure 7a. The relationship of the values sorted during parallel quicksort.
Figure 7b. Computation of the position of each process's comparator value for the final sorted array.

References

Notes for Contributors

The purpose of this Journal will be to publish original papers in any field of computing. Papers submitted may be research articles, review articles, exploratory articles of general interest to readers of the Journal. The preferred languages of the Journal will be the congress languages of IFIP although papers in other languages will not be precluded.

Manuscripts should be submitted in triplicate to: Prof. G. Wiechers at:
Department of Computer Science
University of South Africa
P.O. Box 392
Pretoria 0001
South Africa

Form of manuscript
Manuscripts should be in double-space typing on one side only of sheets of A4 size with wide margins. The original ribbon copy of the typed manuscript should be submitted. Authors should write concisely. The first page should include the article title (which should be brief), the author’s name, and the affiliation and address. Each paper must be accompanied by a summary of less than 200 words which will be printed immediately below the title at the beginning of the paper, together with an appropriate key word list and a list of relevant Computing Review categories.

Tables and figures
Illustrations and tables should not be included in the text, although the author should indicate the desired location of each in the printed text. Tables should be typed on separate sheets and should be numbered consecutively and titled.

Illustrations should also be supplied on separate sheets, and each should be clearly identified on the back in pencil with the Author’s name and figure number. Original line drawings (not photoprints) should be submitted and should include all relevant details. Drawings, etc., should be submitted and should include all relevant details. Drawings, etc., should be about twice the final size required and lettering must be clear and “open” and sufficiently large to permit the necessary reduction of size in block-making.

Where photographs are submitted, glossy bromide prints are required. If words or numbers are to appear on a photograph, two prints should be sent, the lettering being clearly indicated on one print only. Computer programs or output should be given on clear original printouts and preferably not on lined paper so that they can be reproduced photographically.

Figure legends should be typed on a separate sheet and placed at the end of the manuscript.

Symbols
Mathematical and other symbols may be either handwritten or typewritten. Greek letters and unusual symbols should be identified in the margin. Distinction should be made between capital and lower case letters between the letter O and zero; between the letter I, the number one and prime; between K and kappa.

References
References should be listed at the end of the manuscript in alphabetical order of author’s name, and cited in the text by number in square brackets. Journal references should be arranged thus:


Proofs and reprints
Galley proofs will be sent to the author to ensure that the papers have been correctly set up in type and not for the addition of new material or amendment of texts. Excessive alterations may have to be disallowed or the cost charged against the author. Corrected galley proofs, together with the original typescript, must be returned to the editor within three days to minimize the risk of the author’s contribution having to be held over to a later issue.

Fifty reprints of each article will be supplied free of charge. Additional copies may be purchased on a reprint order form which will accompany the proofs.

Only original papers will be accepted, and copyright in published papers will be vested in the publisher.

Letters
A section of “Letters to the Editor” (each limited to about 500 words) will provide a forum for discussion of recent problems.

Hierdie notas is ook in Afrikaans verkrygbaar.
Contents/Inhoud

The Design Objectives of Quadlisp* ........................................3
S W Postma

A CSP Description of some Parallel Sorting Algorithms* ...........7
M H Linck

The Design and Microprogrammed Implementation of a Structured
Language Machine.................................................................13
G R Finnie

Micro-Code Implementation of Language Interpreters* .............19
P P Roets

An Interactive Graphical Array Trace* ....................................23
S R Schach

An Efficient Implementation of an Algorithm for Min-Max
Tree Partitioning.................................................................27
Ronald I Becker, Yehoshua Perl, Stephen R Schach

The Relative Merits of Two Organisational Behaviour Models for Structuring a Management Information System .....................31
Peter Pirow

*Presented at the second South African Computer Symposium held on
28th and 29th October, 1981.