Yugoslav Journal of Operations Research
9 (1999), Number 1, 129-139

COMMUNICATION MODELS: AN EDUCATIONAL FRAMEWORK FOR PARALLEL PROGRAMMING

Miroslav HAJDUKOVIĆ, Branko PERIŠIĆ, Danilo OBRADOVIĆ

Faculty of Engineering, University of Novi Sad
Fruškogorska 11, 21000 Novi Sad,
Yugoslavia

Abstract: This paper presents the abstract representation of parallel computers in the form of communication models. The communication models are intended to be an educational framework for parallel programming, independent of specific details of parallel computer architectures.

Keywords: Parallel programming, parallel computers.

1. INTRODUCTION

Monoprocessor computers show a high degree of uniformity [9]. That is why it is possible to represent monoprocessor computers by one abstract model in the form of procedural high level programming languages [5]. Such a model makes programming transparent to monoprocessor computer architecture details (these details are the concern of compilers, rather than programmers). Unfortunately, parallel computers do not show such uniformity [2]. According to their characteristics, different classifications of parallel computers exist. A well known example of such a classification is the Flynn classification [7] of parallel architectures to MISD, SIMD and MIMD classes. Parallel computers from the MIMD class belong either to multiprocessors or to multicomputers [10]. These two subclasses are important because they influence the style of parallel programming. So, the shared memory style of parallel programming is inspired by multiprocessors, while the message passing style of parallel programming is inspired by multicomputers. The first style is simpler for programmers, but the second style is more general [10].

Serious differences between the architectures of parallel computers have influenced many, very different, high level programming languages, adapted to particular parallel computer architecture [1], [2], [4], [6]. The close adaptation of high

level programming languages to particular parallel computer architectures is a natural consequence of the necessity to use all advances of a particular parallel computer architecture to the greatest possible extent. But, it is possible to make an effort in the opposite direction and devise abstract models of a parallel computer, best suited to some problem classes. Such an effort is especially important for the purposes of parallel programming education, because it eliminates unnecessary details of the functioning of a particular parallel computer architecture, and lets one concentrate only on the essence of parallelism.

The abstract models of a parallel computer could have the form of communicating sequential processes [11]. The exact form of such an abstract model depends on the communication pattern suitable for the optimal cooperation of processes needed to solve some classes of problems. In this paper such abstract models are called communication models, and the typical communication models are presented. Their design has been strongly influenced by gained experience in parallel programming teaching. This experience suggests that parallel programming should benefit as much as possible from (monoprocessor) multiprogramming (it is easier to make the transition from multiprogramming to true parallel programming, than from sequential to parallel programming). For success in parallel programming teaching it is important to offer exact (predefined) forms of parallelism (that lead one to parallel solutions of problems). Such forms of parallelism should be adaptable to different kinds of problems. Also, they should offer natural ways of connecting sequential and parallel parts of devised parallel algorithms. All such forms should be uniform and simple (to be learn and use easily and fast).

The communication models are intended to provide an educational framework for parallel programming. Examples of their usage are shown, and their characteristics and implementation are discussed.

2. COMMUNICATION MODELS

The communication models rely upon processes and process cooperation mechanisms. The processes correspond to independent (parallel) activities. Important properties of processes are their descriptions, as well as their creation and destruction. The processes are identified by identities in the form of unique integers (P_i, $i = 1,2,3,4,\ldots$).

The process cooperation mechanisms are based on mailboxes, and send and receive operations. There are ordinary (o) and special (s) mailboxes. All mailboxes have the same (finite) capacity (so every mailbox accepts a finite number of messages). The send operation places a message into the designated mailbox. The receive operation pulls out a message from the designated mailbox. Both operations are asynchronous [3]. This means that if a mailbox is full, sending blocks the process activity until the mailbox becomes not full. Also, if a mailbox is empty, receiving blocks the process activity until the mailbox becomes not empty.

The formal definition of the communication models is based on syntax of the programming language C.

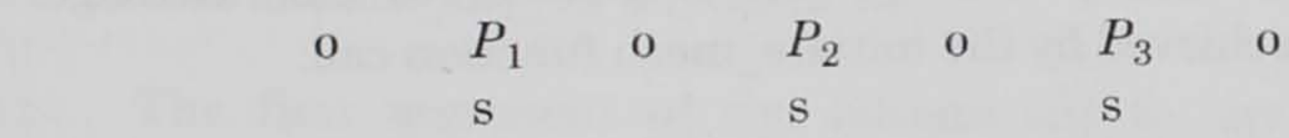
C functions are suitable for the description of processes. The functions which describe processes are of the process type. The create function is intended for process creation. The only argument of a create function call is the address of a process function. Process destruction happens automatically at the end of process function execution.

A communication model program execution begins with the automatic creation of the initial process. Besides that, the initial process function is the same as ordinary process functions. The initial process has higher priority than other processes.

To simplify the communication models' description, the content of messages is limited to integer type values. The arguments of a send function call are a mailbox identity and message content. The only argument of a receive function call is a mailbox identity. Such a call returns the received message content. Mailbox identities are unique integers. Close relationships exist between message boxes and processes. Consequently, process identities are mapped to mailbox identities. Special mapping functions are used for that. The only argument of any mapping function call is a process identity. Such a call returns a mailbox identity.

Four communication models are described: array, mesh, tree and any_to_any.

The array communication model (for 3 processes) may be presented as:



Every process (P_i, $i = 1,2,3$) has two ordinary (o) and one special (s) mailbox. The functions:

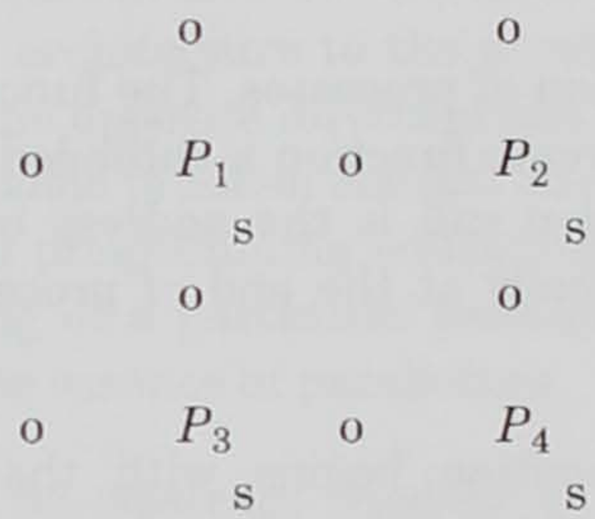
```
int left(int);
int right(int);
int special(int);
```

map a process identity to an identity of its left, right and special mailbox (respectively). Only the initial process can access all mailboxes. All other processes are limited only to their mailboxes. Every mailbox has a capacity of one message. Initialization of all mailboxes is achieved by the initiate_array function call:

```
void initiate_array(int);
```

The only argument of the initiate_array function call is the total number of processes.

The mesh communication model (for 4 processes) may be presented as:



Every process ($P_i, i = 1,2,3,4$) has four ordinary (o) and one special (s) mailbox (the left mailboxes of the processes P_1 and P_3 are the right mailboxes for the processes P_2 and P_4, as well as the upper mailboxes of the processes P_1 and P_2 are the lower mailboxes for the processes P_3 and P_4). The functions:

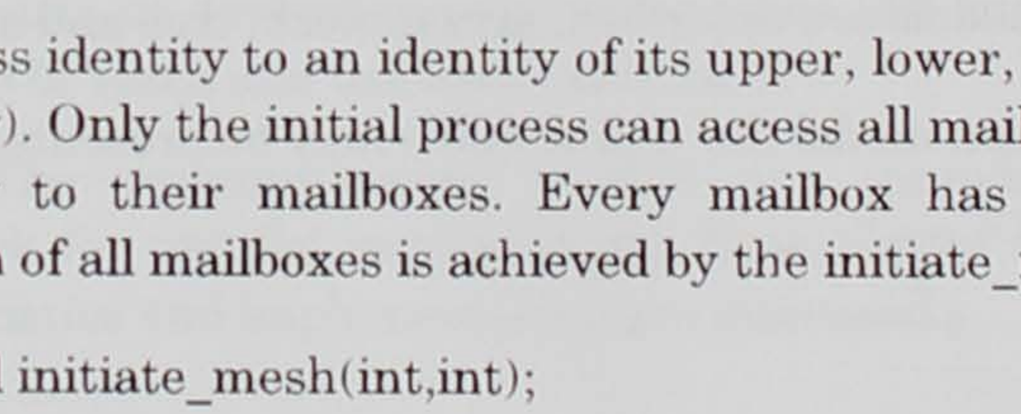
```
int upper(int);.
int lower(int);
int left(int);
int right(int);
int special(int);
```

map a process identity to an identity of its upper, lower, left, right and special mailbox (respectively). Only the initial process can access all mailboxes. All other processes are limited only to their mailboxes. Every mailbox has a capacity of one message. Initialization of all mailboxes is achieved by the initiate_mesh function call:

```
void initiate_mesh(int,int);
```

The first argument of the initiate_mesh function call is the total number of mesh rows, and the second argument is the total number of mesh columns.

The tree communication model (for 3 processes) may be presented as:



Every process ($P_i, i = 1,2,3$) has three ordinary (o) and one special (s) mailbox. The functions:

```
int upper(int);
int lower_left(int);
int lower_right(int);
int special(int);
```

map a process identity to an identity of its upper, lower_left, lower_right and special mailbox (respectively). Only the initial process can access all mailboxes. All other processes are limited only to their mailboxes. Every mailbox has a capacity of one message. Initialization of all mailboxes is achieved by the initiate_tree function call:

```
void initiate_tree(int);
```

The only argument of the initiate_tree function call is the total number of processes.

The any_to_any communication model (for 3 processes) may be presented as:

initial	P_1	P_2	P_3
s	o	o	o

The initial process has a special (s) mailbox. Other processes ($P_i, i = 1,2,3$) have only up to one ordinary (o) mailbox. The process identities are used as the mailbox identities (integer 0 is used as the identity of the special mailbox). All processes can access all mailboxes. Mailbox capacities are defined during mailbox initialization:

```
void initiate_any_to_any(int,int);
```

The first argument of the initiate_any_to_any function call is the total number of processes, and the second argument is the capacity of a particular mailbox.

3. EXAMPLES OF COMMUNICATION MODEL USAGE

The array communication model is suitable for solving the problem of sorting integers (in descending order). Such sorting is achieved if every process keeps the highest received integer, and sends all other integers to its successor:

```
process sorter(int index)
{       int count, old, new;
        count = receive(left(index));
        if ((--count) > 0)
                send(right(index),count);
        old = receive(left(index));
        while ((count--) > 0)
                if ((new = receive(left(index))) > old)
                {       send(right(index),old);
```

```
                old = new;    };
            else
                send(right(index),new);
        send(special(index),old);
};
```

The initial process initiates the array of the mailboxes, creates processes, sends integers 1, 7 and 5 to sorting, takes them from sorting and shows the sorted array:

```
process initial(void)
{       initiate_array(3);
        create(sorter,1); create(sorter,2); create(sorter,3);
        send(left(1),3);        /* number of integers */
        send(left(1),1); send(left(1),7); send(left(1),5);
        show(receive(special(1)));
        show(receive(special(2)));
        show(receive(special(3)));
};
```

The mesh communication model is suitable for solving the problem of matrix multiplication. For example, the product of matrices A and B is:

$$\begin{matrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{matrix} \times \begin{matrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{matrix} = \begin{matrix} a_{11}b_{11}+a_{12}b_{21} & a_{11}b_{12}+a_{12}b_{22} \\ a_{21}b_{11}+a_{22}b_{21} & a_{21}b_{12}+a_{22}b_{22} \end{matrix}$$

Calculation of each of the four elements of the product is independent and can be done by a separate process if it has the necessary elements of matrices A and B. If process P_1 has elements a_{11} and b_{11}, process P_2 has elements a_{12} and b_{22}, process P_3 has elements a_{22} and b_{21}, and process P_4 has elements a_{21} and b_{12}, they can calculate in parallel one-half of the expression describing the corresponding element of the product. To calculate the other half, the processes must exchange elements of matrices A and B. The exchange is regular, so all processes send an element of matrix A right, and an element of matrix B down, and receive a new element of matrix A from the left, and a new element of matrix B from above.

```
process multiplier(int index)
{       int c = 0;
        int i = 2;
        int a,b;
        while ((i--) > 0)
        {       a = receive(left(index));
                b = receive(upper(index));
                c += a*b;
                send(right(index),a);
```

```
                send(lower(index),b);       };
        send(special(index),c);
};
```

The initial process initiates mesh mailboxes, deposits the elements of matrix A:

```
11      12
13      14
```

and the elements of matrix B:

```
21      22
23      24
```

in the appropriate mailboxes, creates processes and shows the elements of the product matrix:

```
process initial(void)
{       initiate_mesh(2,2);
        send(left(1),11); send(left(2),12);          /* a11 and a12 */
        send(left(3),14); send(left(4),13);          /* a22 and a21 */
        send(upper(1),21); send(upper(2),24);        /* b11 and b22 */
        send(upper(3),23); send(upper(4),22);    /* b21 and b12 */
        create(multiplier,1); create(multiplier,2);
        create(multiplier,3); create(multiplier,4);
        show(receive(special(1)); show(receive(special(2));
        show(receive(special(3)); show(receive(special(4));
};
```

The tree communication model is suitable for solving the problem of the parallel summing of integers. All processes in charge of summing add two integers received from below and send up the sum:

```
process summator(int index)
{       int a,b,c;
        a = receive(lower_left(index));
        b = receive(lower_right(index));
        c = a+b;
        send(upper(index),c);
};
```

The initial process initiates tree mailboxes, deposits integers 10, 20, 30 and 40 in the appropriate mailboxes, creates processes and shows the sum:

```
process initial(void)
{       initiate_tree(3);
```

```
        send(lower_left(2),10); send(lower_right(2),20);
        send(lower_left(3),30); send(lower_right(3),40);
        create(summator,1); create(summator,2); create(summator,3);
        show(receive(up(1)));
};
```

The any_to_any communication model is suitable for the parallel finding of distances from a given node to all other nodes in a directed graph (in this example it is presumed that the graph consists of three nodes). All nodes have distances. By definition the distance of the given node is 0. Starting distances for all other nodes are ENDLESS. If a distinct process is assigned to every node, then the requested distances can be found in parallel. To achieve this, every process repeats the same procedure until all the distances are found. In the first step of the mentioned procedure every process sends potentially new distances to its successors (there are at most two successors). In the second step every process receives its potentially new distance from its predecessors (there are at most two predecessors):

```
process node(int index, int old_distance)
{       int new_distance, predecessor_number, successor_number, change, i;
        int succesors[2];
        int distances[2];
        predecessor_number = receive(index);
        successor_number = receive(index);
        for (i = 0;i < successor_number;i++)
        {       successors[i] = receive(index);
                distances[i] = receive(index);      };
        do
        {       change = 0;
                if (old_distance == ENDLESS)
                        for (i = 0;i < successor_number;i++)
                                send(successors[i],ENDLESS);
                else
                        for (i = 0;i < successor_number;i++)
                                send(successors[i],old_distance+distances[i]);

                for (i = 0;i < predecessor_number;i++)
                {       new_distance = receive(index);
                        if (old_distance > new_distance)
                        {       old_distance = new_distance;
                                change = 1;     };
                };
                send(0,change);
        }
        while(receive(index) > 0);
        show_node(index,old_distance);
};
```

The initial process initiates any_to_any mailboxes and deposits in them data such as the number of predecessors, number of successors, successor identities and arch lengths (it is supposed that distances from node 1 to nodes 2 and 3 are to be found and that the arch from node 1 to node 2 has length 10, the arch from node 1 to node 3 has length 40 and the arch from node 2 to node 3 has length 20). After that the initial process creates processes and decides when distances have been found (when there are no changes in the found distances):

```
process initial(void)
{       int change_count,i;
        initiate_ant_to_any(3,6);
        send(1,0);                      /* number of P1 predecessors */
        send(1,2);                      /* number of P1 successors */
        send(1,2); send(1,10);          /* P1 successor index and arch length */
        send(1,3); send(1,40);          /* P1 successor index and arch length */
        send(2,1);                      /* number of P2 predecessors */
        send(2,1);                      /* number of P2 successors */
        send(2,3); send(2,20);          /* P2 successor index and arch length */
        send(3,2);                      /* number of P3 predecessors */
        send(3,0);                      /* number of P3 successors */
create(node,1,0);
create(node,2,ENDLESS); create(node,3,ENDLESS);
do
{       change_count = 0;
        for (i = 1;i < 4;i++)
                change_count += receive(0);
        for (i = 1;i < 4;i++)
                send(i,change_count);
}
while(change_count > 0);
```

4. COMMUNICATION MODEL IMPLEMENTATION

One version of the described communication models is implemented in the programming language conCert (concurrent C for embedded real time) [8]. The programming language conCert is designed for concurrent programming on a monoprocessor computer. Although the programming language conCert supports only semi-parallelism, its multiprocessing environment offers a valuable test bed for the communication models.

5. PROPERTIES OF THE COMMUNICATION MODELS

Parallel programming, based on the communication models, naturally follows from (monoprocessor) multiprogramming, due to the chosen characteristics of communication model elements (their processes and message passing primitives).

The communication models help to visualize typical patterns of parallelism (they do not offer processes and message passing primitives only, but also define their topology). The communication models are adaptable (for example, with no difficulty the planar communication models presented could be accommodated to three-dimensional problems).

The communication models introduce special mailboxes as a natural interface between serial and parallel parts of the same program.

The communication models offer a simple, uniform and abstract framework for parallel programming, therefore they are used with no difficulties.

6. CONCLUSION

Educational reasons were behind the development of the communication models. The properties of the communication models makes them easy to comprehend and use. That is why they are successfully used for laboratory exercises in parallel programming.

The communication models are inspired by CSP [11]. Between CSP and the communication models there are important differences, for example in the characteristics of the communication mechanisms. Besides that, the communication models give precise data about the communication connection of processes. Each particular communication model is influenced by some parallel computer architecture class (for example, array corresponds to systolic array, mesh corresponds to SIMD architecture, tree corresponds to data flow architecture, any_to_any corresponds to hypercube MIMD architecture). Besides the presented, other communication models are possible as well (their definition depends solely on the need to solve some new classes of problems).

REFERENCES

[1] Akl, S. G., *The Design and Analysis of Parallel Algorithms*, Prentice Hall, 1989.

[2] Almasi, G. S., and Gottlieb, A., *Highly Parallel Computing*, Benjamin/Cummings, 1989.

[3] Andrews, G. R., *Concurrent Programming*, Benjamin/Cummings, 1991.

[4] Andrews, G. R., and Schneider, F. B., "Concept and notations for concurrent programming", *ACM Computing Surveys*, 15 (1983) 3-43.

[5] Backus, J., "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs", *Communications of the ACM*, 21 (1978) 613-641.

[6] Bal, H. E., Steiner, A. S., and Tanenbaum, A. S., "Programming languages for distributed computing systems", *ACM Computing Surveys*, 21 (1989) 261-322.
[7] Flynn, M. J., "Very high-speed computing systems", *Proc. IEEE*, 54 (1966) 1901-1909.
[8] Hajdukovic, M., "Concurrent programming in the programming language conCert", University textbook (in Serbian), 1996.
[9] Hayes, J. P., *Computer Architecture and Organization, 2nd ed.*, McGraw-Hill, 1988.
[10] Hennessy, J. L., and Patterson, D. A., *Computer Architecture: a Quantitative Approach*, Morgan Kaufmann, 1990.
[11] Hoare, C. A. R., "Communicating sequential processes", *Communications of the ACM*, 21 (1978) 666-677.