SOFSEM'93 - Hrdoňov, Šumava, Czech Republic, 21.11.-3.12. 1993

The Common Model for the Communication and Synchronization Primitives

Petr Hanáček
Technical University of Brno
Faculty of Electrical Engineering
Department of Computer Science and Engineering
CS-612 66 Brno, Božetěchova 2
email: hanacek@dcse.fee.vutbr.cz

Abstract

The purpose of this work is to present a common model for description of the commonly used primitives for interaction. The attempt is made to uniformly describe the most of common paradigms used in the programming languages for parallel and distributed computing. The shared persistent data space model based on asynchronous PRAM is used as a tool for describing all of the common interaction primitives.

Key words:
parallel computing, communication, synchronization, message passing, shared memory, Linda language.

Introduction

Parallel computing is a popular research area in computer science. Parallel computing differs from the conventional sequential computing so that when in sequential computing only one processor is used in computation, in parallel computing the problem is divided into several subproblems that are solved using a set of processors that work simultaneously, i.e. in parallel.

In parallel computing there is usually very much *interprocess communication* and *process synchronization* because problems must be divided and solutions must be combined and intermediate results must be passed from one process to another. It is clear that interprocess communication, process synchronization and data sharing are probably the most important problems in parallel and distributed computing. In this text we will treat with the communication, synchronization, and data sharing primitives, so we will use the term *interaction* as a common name for these three terms.

The shared data space

For the description of the data space we will use the commonly known asynchronous PRAM computation model. In terms of PRAM model of computation, a *data space,* denoted by DS, is an unlimited set of shared registers of the PRAM model. For our purposes, a content of one PRAM shared register is a bit string of finite, but arbitrary, length. We shall refer to these bit strings as *data objects*. Note, that the length of this bit string is not fixed, it can change in the process of computation. In addition to the bits comprising a data object, each object can also be identified by its ordinal position. In the PRAM computational model the ordinal position is equal to the shared register number. We call this ordinal position a *unique identifier*, denoted *id*.

We would like show in the following text that the commonly used interaction primitives can be modelled using the data space interaction model. In this model the *data space object* is a structure, called *interaction point*. This interaction point consists of two queues - the *entry queue* and the *waiting queue*. These two queues are generally of unlimited length. The entry queue contains the data, which were put into the interaction point by the *out* operations. This queue is a queue of data items, called *elements* that are bit strings of finite, fixed length. The waiting queue contains the processes (or any kind of process descriptors) which have executed the *in* operation and are waiting to completion of this operation.

The two Linda-like operations are associated with this interaction point -- the *in* operation for input and the *out* operation for output. These operations have following notation

$\texttt{in}\ \ (\texttt{id}_i,\ \texttt{P}_j,\ \texttt{D}_l)$
$\texttt{out}\ \ (\texttt{id}_i,\ \texttt{P}_j,\ \texttt{D}_l)$

where this are interpreted as: process P_j does data input from interaction point id_i or data output to object id_i, respectively. The result of the data input is the data value D_l.

The semantics of the *in* operation is following: If the entry queue is not empty, then the first element of this queue is removed from the entry queue. Value of this element is the return value of the *in* operation. If the entry queue is empty, the process P_j is suspended and its descriptor is added at the end of the waiting queue. The *in* operation is blocking. The process performing *in* operation is blocked until the requested data is available.

The semantics of the *out* operation is following: The data element D_l is appended at the end of the entry queue of the interaction point id_i. If the waiting queue is empty, the *out* finished. If the waiting queue is not empty the corresponding *in* operation is completed. The process P_W on the head of the waiting queue is removed from the queue, the data D_W from the head of the entry queue is removed from the entry queue and process P_W performing *in* operation is re-scheduled. The *out* operation is non-blocking, so the process performing the *out* operation immediately continues.

For further definitions will be interesting to know, what processes is the object referenced from. To obtain this information we will define the function

$\texttt{ref}\ (\texttt{id}_i,\ \texttt{DS\_operation})$

that returns a set of processes that contain in their body the *DS_operation* referencing the object id_i. The *DS_operation* can be *in*, *out*, or any DS operation defined later.

Now we can define three important attributes of the interaction point - the functions *elemwidth* (id_i), *maxentry* (id_i), and *maxwaiting* (id_i). The function *elemwidth* returns the numerical value of the size of a data element, that is associated with the interaction point id_i. The functions *maxentry* and *maxwaiting* return the maximal length of the entry (waiting) queue of the interaction point id_i.

In this point we have defined all we need to introduce the *parametrized interaction point*. Parametrized interaction point is an interaction point that is characterized using four attributes: *elemwidth* (id_i), *maxentry* (id_i), *maxwaiting* (id_i), and value *id.init* that is the initial number of elements in the entry queue.

Simple interaction primitives

In this section we try to find relation between above parametrized defined interaction point and real interaction primitives. With a single interaction point we have available only one waiting queue. So we can model only such primitives that use only one queue of waiting processes. The example of such primitive is asynchronous communication channel. This channel has only one kind of waiting processes - the processes waiting for reading. Such primitives we will call *simple interaction primitives*. The opposite case is the *compound interaction primitive*. It is such primitive that needs two (or even more) waiting queues. We will discuss these primitives later.

Now we will try to make a classification of the simple interaction primitives using the parametrized interaction point. We define following three properties D, E, and W of the parametrized interaction points:

D - elemwidth $(id_i) > 0$
E - maxentry $(id_i) > 1$
W - maxwaiting $(id_i) > 1$

Intuitively, the simplest parametrized interaction point (denoted by symbol 0) has no of these three properties, and the most complex (denoted by symbol DEW) parametrized interaction point has all three properties. Using this three properties we can distinguish 8 different simple interaction primitives. The set of these primitives is possible to draw as a lattice diagram - see Figure. 1.a. For better understanding is possible to denote the elements of this lattice by the common used names of the simple interaction primitives. In the picture are used following names: asynchronous mailbox with unlimited capacity (*AsMailb*), asynchronous channel with unlimited capacity (*AsChan*), semaphore (or monitor) (*Sem*), and shared variable (*Var*).

```
               DEW                           DEW
             AsMailb                         ???
               /|\                           /|\
             /  |  \                       /  |  \
           /    |    \                   /    |    \
         /      |      \               /      |      \
      DW        DE       EW         DW        DE       EW
     Var2     AsChan     Sem4      Farm      RPC2    SynMailb2
      |\       / \       /|         |\       / \       /|
      |  \   /     \   /  |         |  \   /     \   /  |
      |    X         X    |         |    X         X    |
      |  /   \     /   \  |         |  /   \     /   \  |
      |/       \ /       \|         |/       \ /       \|
     Var1      Sem2      Sem3      RPC1    SynMailb1 SynMailb3
      D\        W        /E         D\        W      Barrier
         \      |      /               \      |      / E
           \    |    /                   \    |    /
             \  |  /                       \  |  /
               \|/                           \|/
a)            Sem1                 b)     SynChan
                0                             0
```

Figure 1: Lattice diagram of simple and compound interaction primitives

From the implementational point of view it is important to determine the maximal value of the *elemwidth*, *maxwaiting*, and *maxentry* functions for the particular interaction point. If some of these maximal values is 0 or 1, we have the possibility to omit one of the interaction point queues. Even if the value of the entry queue or the waiting queue is not equal to 1 but is limited, information about the maximal lengths of these queues can help for example in space allocation and can improve the performance of particular implementation of the interaction point.

Obtaining the *elemwidth* is quite straightforward. It can be done using the static analysis of the source code. But the obtaining the *maxentry* and *maxwaiting* values is not so simple. We will present the formulas for computing the *maxentry* and *maxwaiting* values, but because of lack of space we will present the formulas without proof. It can be proven that for simple interaction primitive the values of the functions *maxwaiting* and *maxentry* are following:

$$\mathrm{maxentry}\ (\mathrm{id}) = \mathrm{id.init} + \sum_{P_j \in \mathrm{ref(id,\ out)}} U_o\ (P_j,\ \mathrm{id})$$

$$\mathrm{maxwaiting}\ (\mathrm{id}) = \sum_{P_j \in \mathrm{ref(id,\ in)}} U_i\ (P_j,\ \mathrm{id}) - \mathrm{id.init}$$

The functions U_i(P,id) and U_o(P,id) are so called *unbalance* functions. These functions, intuitively saying, express "the contribution of process P to demand for length of the entry queue or waiting queue, respectively". For example if the execution thread of the process contains only one *in* operation followed by one *out* operation, then the unbalance values are U_i=1 and U_o=0. The basic formulas for computing the unbalance functions are following:

$$U_o\ (P,\ \mathrm{id}) = \underset{*}{\mathrm{MAX}}\ \left(\mathrm{MAX}_{i=1}^{k}\ \left(\sum_{j=1}^{i} \begin{cases} -1 & \text{if } op_j = \text{in} \\ +1 & \text{if } op_j = \text{out} \end{cases}\right)\right)$$

$$U_i\ (P,\ \mathrm{id}) = \underset{*}{\mathrm{MAX}}\ \left(\mathrm{MAX}_{i=1}^{k}\ \left(\sum_{j=1}^{i} \begin{cases} +1 & \text{if } op_j = \text{in} \\ -1 & \text{if } op_j = \text{out} \end{cases}\right)\right)$$

* maximum through all operator sequences $op_1 \ldots op_k$ with interaction point *id* in process P

The above formulas are correct but practically useless. It is computationally infeasible to compute the first MAXimum in the formulas because this maximum is made through all operator sequences $op_1 ... op_k$ (execution threads) with interaction point *id* in process *P*. Generally, we are not able to determine in compile-time analysis the number of execution of the program loops so the number of all execution threads is unlimited.

One of the solution of this problem is following: We transform the source code of the process into the control-flow graph. On this graph we perform so called *invariant transformations*. An invariant transformation of the control flow graph is such graph transformation that preserve the values of unbalance functions. The problem of computation of the unbalance functions is not completely solved yet but because of existence of invariant transformations it is clear that it is possible to solve it.

Compound interaction primitives

We have considered in previous paragraphs only this kind of interaction primitives that consists from a single interaction point. Now we will consider the interaction primitives that consist of exactly two interaction points. These primitives we will call *compound interaction primitives*.

Similarly as in case of the simple interaction point we will try to make a classification of the compound interaction primitives. We define three properties D, E, and W of the compound interaction primitive:

D - $(\mathrm{elemwidth}\ (id_1)>0)$ and $(\mathrm{elemwidth}\ (id_2)>0)$
E - $\mathrm{maxentry}\ (id_1) = \mathrm{maxentry}\ (id_2) > 1$
W - $\mathrm{maxwaiting}\ (id_1) > 1$

Using these three properties we can distinguish 8 different compound interaction primitives. The set of these primitives is possible to draw as a lattice diagram - see Figure. 1.b. In this picture are used names for following common simple interaction primitives: synchronous mailbox with limited capacity (*SynMailb*), synchronous channel with limited capacity (*SynChan*), Remote Procedure Call (*RPC*), farm of processes (*Farm*), and barrier synchronization operation (*Barrier*).

It can be proven that for compound interaction primitive the values of the functions *maxwaiting* and *maxentry* are following:

$$\texttt{maxwaiting}\ (id_1) = |\texttt{ref}\ (id_1, \texttt{in})|$$
$$\texttt{maxwaiting}\ (id_2) = |\texttt{ref}\ (id_2, \texttt{in})|$$
$$\texttt{maxentry}\ (id_1) = \texttt{maxentry}\ (id_2) = \sum_{P_j \in \texttt{ref}(id_1, \texttt{out})} U_o\ (P_j, id_{12}) + \sum_{P_j \in \texttt{ref}(id_2, \texttt{out})} U_o\ (P_j, id_{12})$$

The evidence of above formulas is beyond scope of this document and is not presented here.

Conclusion

As a most important new idea of the work it is considered the design of the interaction point together with building of the common model and classification of the simple and compound interaction primitives. Another contribution is an analyse of implementational optimization of this model, mainly formulas for computing the *maxentry* and *maxwaiting* functions. The developing of an algorithm for computing the unbalance functions U_i and U_o was described only briefly. The work on this algorithm is not finished yet.

References

[1] Pfaltz, John L.: Programming over a Persistent Data Space, IPC Technical Report 92-008, University of Virginia, 1992

[2] Williams, Shirley A.: Programming models for parallel systems, John Wiley & Sons 1990, ISBN 0 471 92304 4