Proceedings of the IFIP 10.3 Conference on Programming Environments for Massively Parallel Distributed Systems, Monte Verita, Ascona, Switzerland, April 1994, K.M. Decker and R.M. Rehmann editors, Birkhauser publisher, pages 365-371

A Library for Coarse Grain Macro-Pipelining in Distributed Memory Architectures

F. Desprez*

Computer Science Department
The University of Tennessee
107 Ayres Hall, Knoxville
TN 37996-1301, USA

Laboratoire LIP
URA 1398 du CNRS
ENS LYON, 46, Allée d'Italie
69364 LYON Cedex 07, France

1 Introduction

The most natural way of programming parallel machines is data-parallelism where programs execute phases of computations and communications on different sets of data and no overlap exists between communications and computations. Moreover, communication phases are synchronous, i.e. every processor executes these phases at the same time and waits until the last processor completes his communication phase. From the perspective of program correctness, these data-parallel programs are much more easier to prove than asynchronous CSP-based parallel programs [Hoa85]. Unfortunately, performances of such programs are bounded by the communications cost. To avoid this problem, the solution is to overlap communications by computations. This is not always possible because of the dependences within the code. If data dependences prevent the use of simple overlap, a solution consists in using a pipelined data parallel algorithm by decreasing the grain of computations, and overlapping communications of previously computed parts by the computation of next parts of the set of data. This is also called coarse grain macro-pipelining [Kin88, Tse93]. The resulting code is then much more efficient but, unfortunately, much more complicated too. Our aim is thus to provide one set of high level routines which implement pipelined data parallel algorithms in the most transparent way to the user of parallel message passing machines.

The LOCCS library (Low Overhead Communication and Computation Subroutines) has been first presented in [DT93] and subsets are now available for the iPSC/860 and Paragon machines.

*This work was supported in part by the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615, CNRS-NSF grant 950.223/07, PRC C^3, Archipel SA and MRE under grant 974, and the DRET.

The LOCCS library is presented as well as an example of its use. Some experimental results on the iPSC/860 and Paragon machines are then reported and discussed. Finally, we describe the one direct application of macro-pipelining using the LOCCS routines.

2 Statement of the problem and macro-pipelining solution

Because of dependences between computations and between communications and computations, overlap of communications is sometimes difficult to obtain. Consider the dependence graph given on Figure 1a. This is a classical graph for pipelined operations. Processor P_i has to compute J_B before sending the result to processor P_j which in turn executes J_A. Because of the dependences between the computation of J_B, communication and computation of J_A, one may think that no overlap is possible. Using coarse grain pipelining, we are able to solve this problem efficiently. If this is possible, i.e. if no vertical dependences prevent from this operation, a solution consists in decreasing the grain of J_B and sending the results of the small j_b "as soon as possible", i.e. after their computation (Figure 1b). The computation of the next parts of J_B is overlaped by the communication the previously computed ones. One the other side of the macro-pipeline, P_j does exactly the same, breaking J_A in parts (j_a), corresponding to the received data. In such algorithm, we have an overlap of communication and a kind of "load balancing" of computations leading to a more efficient algorithm.

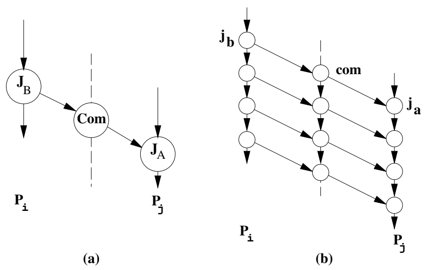

Figure 1: One dependence graph and is macro-pipeline solution.

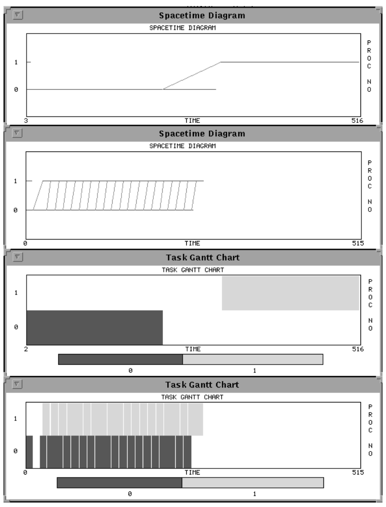

Figure 2: Execution of a classical macro-pipeline on Intel iPSC/860.

On Alg. 1, initial programs of P_i and P_j are given. D denotes the data which have to be communicated between the two processors after a computation phase on P_i (J_B) and before another computation phase on P_j (J_A). Alg. 2 shows the macro-pipeline version of the program executed by processor P_i.

J_B(D, size_of_D)	receive(D, size_of_D)
send(D, size_of_D, P_j)	J_A(D, size_of_D)

Alg 1: Programs of processor P_i and P_j.

```
jb (0, ν, subblock of D)                              } Initialization
for i=0 to number_of_packets-2                        \
   in // do                                           |
      send(subblock i of D, ν, Pj)                    |  Normal reg.
      jb (i+1, ν, subblock of D)                      |
   enddo                                              |
endfor                                                /
send(subblock number_of_packets-1 of D, ν, Pj)        } Terminaison
```

Alg 2: Coarse grain pipelining for processor P_i.

The trace of the execution of such algorithm on a iPSC/860 with the same tasks before and after the communication is presented on Figure 2 using Paragraph Space Time and Task Gantt views [HE91]. We have a gain of a factor almost equal to 2 that makes the overlap almost optimal. The Gantt views show the tasks and micro-tasks, used in the macro-pipelining execution. The size of the packet ν can be computed theoreticaly and experimentaly. This is the most important factor since it directly affects the performance results. If the number of packets is too large, the cost of the startup times is also too large, leading to poor performance. On the other hand, if the number of packets is too small, the overlap is not complete. Thus, there is an optimal packets size which needs to be carefully determined.

3 The LOCCS library

We have designed and implemented a general library for coarse grain pipelining on different parallel machines. There are 8 routines, corresponding to the classical communication patterns.

Routine	Communication pattern
`loccs_oto`	One-to-one
`loccs_exchange`	Exchange between 2 processors
`loccs_shift`	Shift
`loccs_ota`	Broadcast
`loccs_pota`	Scattering
`loccs_ato`	Gathering
`loccs_ata`	All-to-all
`loccs_pata`	Multi-scattering

Table 1: LOCCS routines and communication patterns.

For each routine, the communication phase is a call to a non-blocking high-level communication routine on a packet of size ν. This ensures the portability of the routine. The user, who supplies the code of the tasks, do not have to know the underlying algorithm used for communication.

We now give an example of a routine, `loccs_oto`, which solves the previous problem of a communication between two processors. The two processors involved can have two tasks to perform, one before and one after the communication. A call to the `loccs_oto` routine is realized by:

`loccs_oto(me, oto, buffer,` ν`, size,` job_b`,` job_a`)`

We have different structures to define environment and buffers. `me` (Id of the caller `loccs_oto`), `oto` (structure giving the id of the two processors), `buffer` (structure for message buffers), ν (packet size), `size` (total size of the message) and job_b, job_a (tasks which will be executed before and after a communication step).

One interest of this library is its portability. We have chosen to implement one subset of the routines on different machines and environments. The first version of the library is written in C but the C++ and Fortran versions are currently under development. Table 2 presents the name of the routines implemented on different machines. For the Intel Paragon, it will be possible to implement the whole set of routines, as soon as we will be able to program the second i860, dedicated to communications.

Architecture	Available routines	Environment used
Intel iPSC/860	`loccs_oto, loccs_ota, loccs_exchange, loccs_shift`	NX/PICL
Intel Paragon	`loccs_oto, loccs_ota loccs_exchange, loccs_shift`	NX and PVM 3.*
Clusters of workst.	`loccs_oto`	PVM 3.*

Table 2: Different implementations of LOCCS routines.

We are now going to describe some experiences performed on the Intel iPSC/860 and Paragon machines. The routine tested is `loccs_oto`. The curves obtained give an idea of the executions of the routines of "real" applications and of the computation of the optimal packet size. The tasks use loops of additions and we make them vary to understand the impact of the machine and algorithm parameters on the execution time. On Figure 5, we change the ratio between the two tasks (before and after communication) for the `loccs_oto` on the iPSC/860 and Paragon machine. The total size of computation is 30 units, where one unit is an addition on each element of the current packet. We have executed the routine for the ratios j_b/j_a equal to 10/20, 15/15 and 20/10. For each ratio, we vary the packet size, from one element to the total size of the message. The "hills" on the curves are due to the size of the last packet which size is in the range [1, total size of the message -1]. We can see that the best execution time is obtained with tasks of equal sizes. On the left of the curve, we have only one packet and no overlap. As the packet size decreases, we have more and more overlap until reaching a minimum. After this minimum, the number of packets is too large and we have an extra overhead due to the message latency. The optimal size of packets is always in the same range ([200..500] for a total size of 10000). This size can be computed very accurately using machine parameters and the costs of the different tasks.

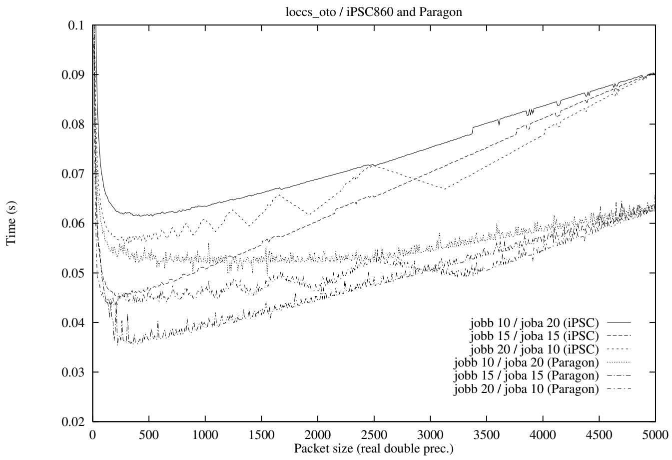

Figure 3: Some executions of `loccs_oto` on Intel iPSC/860 and Paragon machines.

4 Application example

In this section, we present one example of application of the LOCCS routines, namely the LU factorization of a dense matrix A which transforms this matrix in the product of two triangular matrices L and U, respectively lower (L) and upper (U) triangular. For sake of simplicity, we describe how to improve a row version, described in [GR88]. Although his method does not lead to the bes performances on DMPC, it is one of the simplest to describe. Macro-pipelining methods can also be used with other storages. Alg. 4 presents the classical *broadcast-cube* algorithm on an hypercube.

```
for k=0 to n-1
    root = calc_root(k)
    Determine pivot row
    Update permutation vector
    /* computation part of the algorithm */
    if (me == root) broadcast pivot row
    else receive pivot row
    /* update part of the algorithm */
    for (all rows  i > k that I own)
            l_ik = a_ik / a_k
            for j=k+1 to n-1
                a_ij = a_ij - l_ik * a_kj
            endfor
    endfor
endfor
```

Alg 4: Parallel LU factorization algorithm on hypercube.

The LOCCS routine `loccs_ota` can be directly used for the LU factorization. On Alg. 4, the part of the algorithm corresponding to broadcast of the pivot row and the update of the remaining rows can be replaced by a single call to the `loccs_ota` routine (Alg. 5).

```
for k = 0 to n - 1
    ν = calc_nu_opt(k)
    root = calc_root(k)
    determine pivot row
    update permutation vector
    loccs_ota(me, ota, bufferota, ν, N-j-1, NOJOB, UPDATE)
endfor
```

Alg 5: Parallel LU factorization algorithm using LOCCS routine `loccs_ota`.

Notice that, using this algorithm for he LU factorization, there is no j_a. Therefore, `NOJOB` is given as a parameter instead of the routine j_a. `UPDATE` performs the update of all the remaining columns using the received packet of size ν. The resulting code is simpler and more efficient.

5 Conclusion and future work

The use of routines like the LOCCS prevents the developer of a parallel program from the redesign of a complicated pipeline program in order to increase the efficiency. Moreover, the code obtained is much more simpler and efficient. The routines can be optimized for each distributed memory machines and each topology. These routines correspond to the SPMD-like paradigm of execution.

Our current work is to complete the implementation of the set of routines on the Intel Paragon, the ARCHIPEL Volvox IS-860 machines and heterogeneous environment (using PVM) and to propose routines for the computation of ν for some classical algorithms. We investigate the possibility of computing ν during the execution to ensure a better load balance (adaptive ν). Subsets of the routines are available on the two Intel machines. We also study an object oriented interface for this library. The LOCCS routines can also be called by a parallel compiler of FortranD-like programs which knows the parameters of the target machine and the data dependences within the code.

References

[DT93] F. Desprez and B. Tourancheau. LOCCS: Low Overhead Communication and Computation Subroutines. In *High Performance Computing and Networking Conference - Amsterdam*. Elsevier, May 1993.

[GR88] G.A. Geist and C.H. Romine. LU Factorization Algorithms on Distributed Memory Multiprocessor Architectures. *SIAM Journal on Science and Statistical Computing*, 4:639–649, 1988.

[HE91] M.T. Heath and J.A. Etheridge. Visualizing Performances of Parallel Programs. Technical Report ORNL/TM-11813, Oak Ridge National Laboratory, July 1991.

[Hoa85] C.A.R. Hoare. *C.S.P.: Communicating Sequential Processes*. Prentice Hall, 1985.

[Kin88] C.T. King. *Pipelined Data Parallel Algorithm: Concept, Design and Modeling*. PhD thesis, Michigan State University - Department of Computer Science, 1988.

[Tse93] C.W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, January 1993.