Algorithm Development for Distributed Memory Multicomputers Using CONLAB

PETER JACOBSON, BO KÅGSTRÖM, AND MIKAEL RÄNNAR

Institute of Information Processing, University of Umeå, S-901 87 Umeå, Sweden

ABSTRACT

CONLAB (CONcurrent LABoratory) is an environment for developing algorithms for parallel computer architectures and for simulating different parallel architectures. A user can experimentally verify and obtain a picture of the real performance of a parallel algorithm executing on a simulated target architecture. CONLAB gives a high-level support for expressing computations and communications in a distributed memory multicomputer (DMM) environment. A development methodology for DMM algorithms that is based on different levels of abstraction of the problem, the target architecture, and the CONLAB language itself is presented and illustrated with two examples. Simulation results for and real experiments on the Intel iPSC/2 hypercube are presented. Because CONLAB is developed to run on uniprocessor UNIX workstations, it is an educational tool that offers interactive (simulated) parallel computing to a wide audience.

1 INTRODUCTION

Today, much algorithm design for parallel computer architectures and most implementations are done in conventional programming languages like Fortran and C. Normally, this is a very time-consuming process, especially in an innovative phase where different ideas and prototype implementations are examined. It would be desirable to be able to express the computations in as high level of abstraction as possible and to focus on the parallelization issues and problems for different architectures. The application area we have in mind is matrix computations that are basic in most scientific, economic, and engineering applications. It is well known that block algorithms are amenable for many parallel architectures [1–3]. Good interactive development environments for sequential computations exist, notably MATLAB (MATrix LABoratory) [4]. MATLAB has simple operators and built-in functions for matrix addition and multiplications, matrix factorizations, etc. This paper presents the CONLAB environment with focus on algorithm development for distributed memory multicomputers (DMM).

CONLAB (CONcurrent LABoratory) is an environment for developing algorithms for parallel computer architectures and for simulating different parallel architectures. This means that the user can experimentally verify and obtain a good picture of the real performance of a parallel algorithm executing on a simulated target architecture. The aim with CONLAB is at least twofold, namely, to provide an environment for developing and testing parallel algorithms, and an educational tool for introducing parallel computing in both teaching and research. Because CONLAB is

Received May 1992
Accepted November 1992

Scientific Programming, Vol. 1, pp. 185–203 1992
CCC 1058-9244/93/020185-19

developed to run on uniprocessor UNIX workstations it offers interactive (simulated) parallel computing to a wide audience. At present, CONLAB is mainly focused on algorithm design for, and simulation of, DMM architectures with message passing communication.

CONLAB gives a high-level support for expressing computations and communications in a DMM environment. Section 2 gives an introduction to the CONLAB language, the simulation of parallel execution in CONLAB, and different tools related to parallel algorithm design and simulation of DMM architectures. Section 3 introduces a development methodology for DMM algorithms that is based on different levels of abstraction of the problem, the target architecture, and the CONLAB language itself. In section 4, the development methodology using CONLAB is illustrated with two examples. Simulation results for and real experiments on the Intel iPSC/2 hypercube are presented in section 5. Finally, in section 6 some conclusions are given and the future development of CONLAB is discussed.

2 THE CONLAB ENVIRONMENT

CONLAB has inherited many characteristics from MATLAB [4]. It is an interactive environment for scientific and engineering computations that also offers parallelism. The language used for programming in CONLAB is similar to the MATLAB language (in reality a subset because all facilities of MATLAB are not implemented) and is extended with constructs used for expressing parallelism, synchronization, and communication. In the discussions and descriptions that follow we assume that the reader has some familiarity with, and experience in, using MATLAB [4] or a MATLAB-like environment. As in MATLAB the only data structure are matrices (scalars are 1×1, vectors are $1 \times n$ [row vectors] or $m \times 1$ [column vectors], and matrices, e.g., $m \times n$). The colon notation is used to specify columns, rows, or submatrices of a matrix A. For example, $A(:, i)$ denotes the i:th column of A and $A(i:j, k:l)$ denotes the entries of A in rows i through j and columns k through l. We will frequently refer to and use concepts like predefined and user-defined functions, different matrix operations, etc.

Because CONLAB is an interactive environment it frees the user from issues like compilation, linking, etc. The user can define a function interactively and call it immediately. In the same way, a process (see section 2.2) can be defined, assigned to a set of processors, and simulated parallel execution can be started instantaneously.

In the following we give an introduction to the parallel concepts of the CONLAB language, the simulation of parallel execution in CONLAB, and different tools related to parallel algorithm design and simulation of DMM architectures. Illustrations of their use are shown in two examples of section 4. Other reports provide more information about CONLAB [5–7].

2.1 Where Does CONLAB Run?

CONLAB is mainly intended to run on uniprocessor UNIX workstations, but it could profitably be ported to multiprocessor workstations or even supercomputers to achieve better performance of big simulations. The computational part of CONLAB is based on the high-performance linear algebra package LAPACK [3], which is portable to many of today's advanced computer architectures. In fact, by running CONLAB on a very powerful machine it is possible to simulate the actual computations faster than running the same application on the DMM target machine (at least for the first and second generations DMM architectures).

2.2 Extensions for Parallelism

Parallelism is expressed with a new control structure, the process. A **process** is very similar to a **function**, in that it is a named collection of statements that can be initialized with a set of arguments. A process can be assigned to one or more virtual processors, and parallel execution of these processes can be simulated by time sharing on a UNIX workstation. Arguments can be passed to the process when it is assigned and they are typically used for different initializations.

Communication via message passing is achieved with **send** and **receive** primitives. A message can be sent to one or more processes and a type is specified for the message. Reception of messages can be done by specifying the sender and/or message type. Communication can be either synchronous or asynchronous. When synchronous communication is used the sender waits until the message is completely received by the receiver, whereas for asynchronous communication the sender continues execution immediately after submitting the message.

2.3 Simulation of Parallel Execution

The simulation in CONLAB is based on a stack machine. The CONLAB statements are compiled to a pseudo code that is executed by a virtual stack machine. The values on the stack are matrices, which are the only data type in CONLAB, and the machine instructions perform matrix operations on the stack elements.

The processes share the CPU of the simulator and execute a series of stack machine instructions operating on their private high-level stack.

To control the simulated parallel execution of the processes, all processes are provided with a time value, which describes how far in execution a process has reached. These values are used to determine the scheduling of the processes as well as to control the synchronization of the processes. For each stack machine instruction a process executes, its time value is increased according to a time model, described in section 2.4.

Parallel execution of processes is achieved by letting the processes execute in a time shared manner. During one time slice a process is allowed to execute a few stack machine instructions, operating on its own private stack, as long as it does not execute instructions with side effects (e.g., communications). Because a side effect is dependent on and changes the global state of the system of processes, a process that executes an instruction with side effects must be the process with the lowest virtual (simulated) time value. Otherwise it has to wait for the other processes "catching up" in virtual time [7]. After one time slice another process is selected for execution, and the new active process operates on its private stack, and so on.

After each instruction a process executes, a new time value is computed for the process. This time value is checked against the time slice and the process is allowed to continue execution as long as the time value does not exceed the time slice.

At each step the process with the smallest time value is selected for execution. This ensures that all processes will eventually be allowed to execute, provided that they perform floating point operations and/or communication.

2.4 The Time Model for Simulation of DMM Architectures

The execution of the processes on the virtual processors is controlled by a time model that computes the elapsed time on each virtual processor. The model is divided into two parts, computations and communications. Different DMM architectures can be simulated by changing this time model.

The time model for computations is based on the number of floating point operations (flops) a process performs, and the cost for one flop. In this context multiplications as well as additions, subtractions, and divisions are considered as floating point operations with the same cost. The user specifies with a call to the function *floptime*, the cost for one floating point operation. Typically, this value represents an average of the practical performance of the node processor. Each time a computation is performed by a process the new time value, *tval*, of that process is computed as

$$tval = tval + \#flops \cdot t_{flop},$$

where t_{flop} is the time cost associated with a floating point operation.

The time model for communications on DMM architectures is based on the fact that on a real DMM, communication can take different scenarios depending on the relative order in which the sending and receiving nodes issue the **send** and **receive** calls, respectively.

1. Unbuffered communication is the result when the receiving node has issued a receive call before the message has arrived at the node. When this happens the communication system knows where to put the message and no communication buffers have to be used.
2. Buffered communication occurs when the receiver has not performed the receive call when the message arrives. Because the communication system cannot know where in the user data area the message is to be placed it is forced to use internal buffers.

The communication time model has the following components:

1. *Send time* is the time it takes for the sender of the message to contact the local communication system. The send time includes copying the message from the local data area to the communication system buffers. The send time is divided into buffered send time and unbuffered send time to distin-

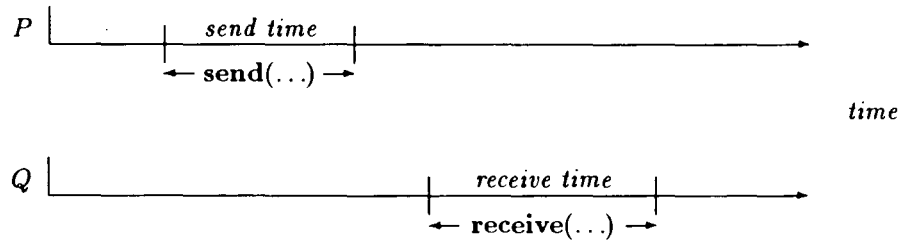

FIGURE 1 Communication time components for buffered communication.

guish between the two types of communication.

2. *Receive time* is the time it takes for the receiver to deal with a message that has arrived at the node. Note that receive time only concerns buffered communication.
3. *Delay time* is the time it takes for the message to reach the target node after the sender has initiated the send operation.
4. *Total communication* time. When unbuffered communication is used the receiver is blocked on receive when the message arrives. The total communication time for unbuffered communication is the time it takes for the message to be fully transmitted to the receiver from the point where the sender issued the **send** call.

The following small program illustrates the communication time components.

```
process P
      ...
      send(message to Q);
      ...
end
process Q
      ...
      receive(message from P);
      ...
end
```

The process P sends a message to the process Q, using the **send** statement. Process Q receives the message from the process P, with the **receive** call.

If Q initiates the **receive** after P has made the call to **send** then buffered communication is used. Figure 1 shows the relationships between the time components when buffered communication is achieved in the above program.

If the **receive** call is executed before the call to **send** then unbuffered communication is used, which is depicted in Figure 2. In Figure 2 the delay time decides when the receiver changes state from idle, waiting for a message to active in communication. During simulation the delay time is used to determine if the message has reached the receiving node by the time the receive call is done. This information is used to decide if buffered or unbuffered communication is to be accomplished.

The communication time components are presented to the CONLAB simulator by specifying a function expressed in CONLAB notation. This function is implicitly called every time a communication is initiated by a **send** call and it is responsible for computing the time components described above. The function takes the message size and type, and the sender and receiver processes as arguments and delivers the send, receive, delay, and total communication times as results. The time components are then used to calculate new time

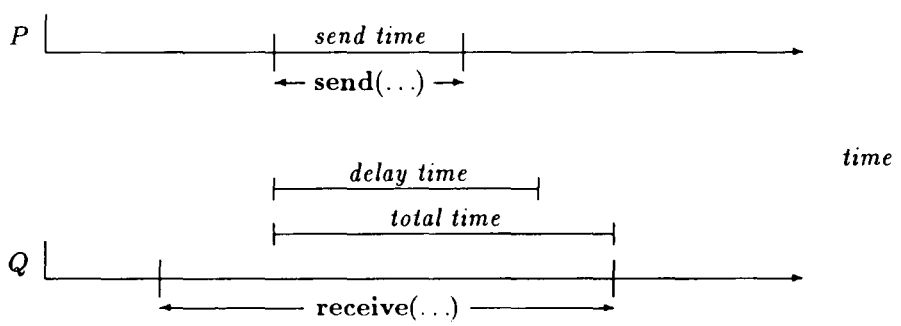

FIGURE 2 Communication time components for unbuffered communication.

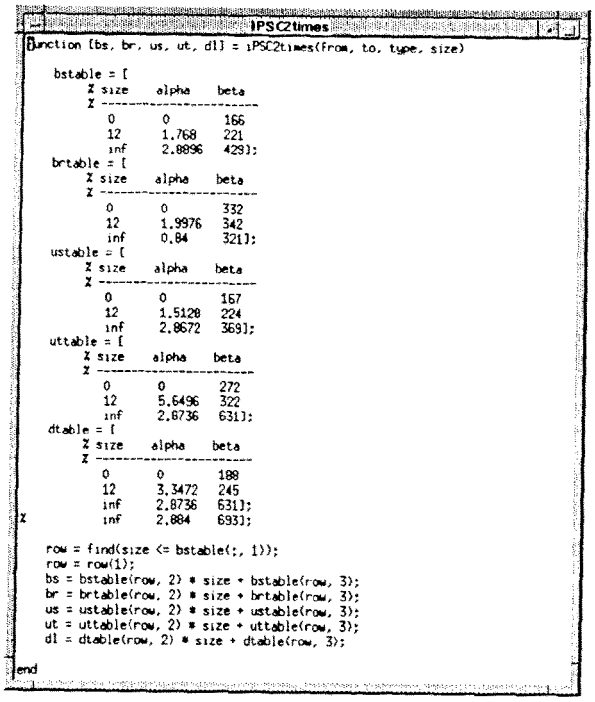

FIGURE 3 CONLAB function for computing communication time components for the iPSC/2 hypercube.

values for the processes involved in the communication.

Figure 3 shows an example of a CONLAB function that computes the communication time components of the Intel iPSC/2 hypercube. The time components for buffered communication (**bs**, **br**) and unbuffered communication (**us**, **ut**, **dl**) are approximated by the usual linear model

$$t_{\text{component}} = \alpha \cdot M + \beta.$$

where β and α denote the start-up cost and the per-unit cost for transferring a message of size M double words (8 bytes), respectively. Detailed communication benchmarks resulting in this time model for communication is described in [8].

By changing the value of *floptime* and/or the tables of the communication time function, the user can, for example, examine different computation-to-communication ratios of a DMM model. Different DMM architectures can be simulated by writing new functions for computing communication time components. The time model for computations does not incorporate the simulation of memory hierarchies. It would then be necessary to extend *floptime* to a function that approximates different components as in the communication time model (e.g., cache effects, pipe-lining).

2.5 Performance Measuring

The functions *timer*, *arithtime*, and *commtime* are used to calculate the timing characteristics of a CONLAB program. The *timer* function returns the current time value of a process (it is initialized to zero when the process starts) and it can be used to measure the elapsed execution time of a process. Similarly, the functions *arithtime* and *commtime* return the current times for arithmetic computations and communications, respectively (both initialized to zero when the process starts).

The arithmetic time can also be computed as *floptime* · *flops*, where the function *flops* returns the number of floating point operations performed by the process.

2.6 Monitoring Process Status During Simulation

During simulation the user can obtain information about the status of all processes. Process utilization (the number of busy processes) and process status (busy or waiting) can be plotted as a function of time (see Fig. 4). Processes in busy state are marked in black (green on a color screen), while processes in waiting or idle states are marked in white (red).

2.7 Animated Replay of Messages

As an option CONLAB will generate trace files of all communications and can therefore offer the user a way of replaying the simulation in terms of communication by message passing. This replay shows sending and receiving processes, respectively, for each message that was transferred. It also shows the message type and size and the times the message was sent and received. The contents of the message can also be printed out. In this way, the message replay is a tool for debugging the communication of a distributed algorithm. Before the replay starts, the user has the option to sort the messages by send time or receive time. The message replay window is shown in Figure 5. This window dump shows that node 11 is receiving a message (of type 2 and length 256 bytes) from node 8.

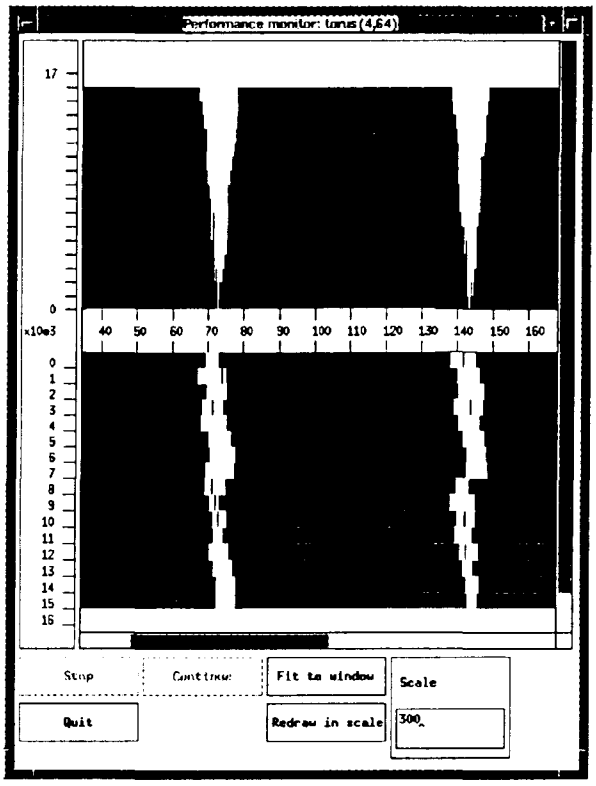

FIGURE 4 The performance utilization (*perf*) window.

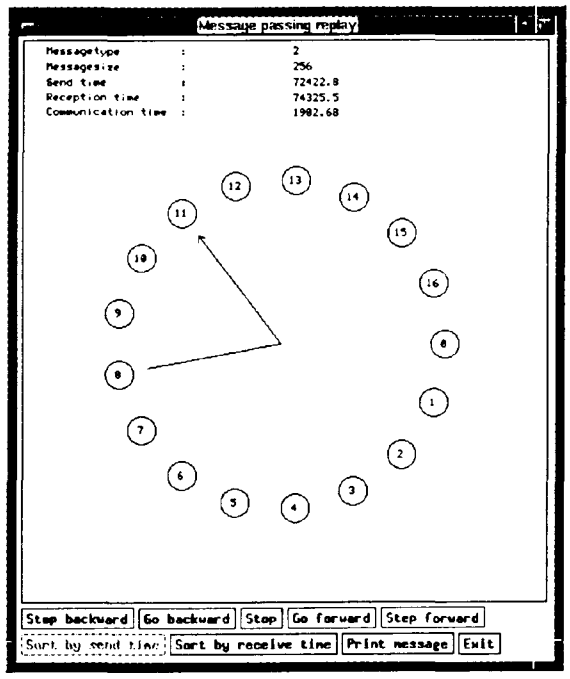

FIGURE 5 The message replay (*replay*) window.

2.8 ParaGraph

There is a possibility to use trace files generated by CONLAB in the visualization tool ParaGraph [9]. ParaGraph, which primarily is intended as a postprocessor to the instrumentation package PICL [10], includes several graphs describing algorithm performance when used with CONLAB.

3 ALGORITHM DEVELOPMENT METHODOLOGY

The methodology for developing parallel algorithms in CONLAB is based on different levels of abstraction of the problem, the target parallel architecture, and of the CONLAB language itself. By following the methodology the user will implement, at as high abstraction level as possible, functions and processes in CONLAB that define the architecture topology, communication on the topology, node and host algorithms, and a user interface. Below, we describe the different levels of abstraction.

1. The first level of abstraction concerns *process topology functions*. These functions describe the topology of the target architecture (real or virtual) and are used to navigate in the topology. For example, process topology functions in a two-dimensional mesh translate a two-dimensional processor index into a single process number and vice versa. Typically, a binary reflected Gray code numbering of the processors in a topology (e.g., ring, mesh, or hypercube) is used to assure that logical adjacent processors are also physical neighbors. Other topology functions may deliver the physical neighbors in the four directions of a two-dimensional mesh (north, south, east, and west).
2. The second level concerns *communication functions*. These functions are problem dependent but can of course be general. Typically, they perform some kind of compound communication on the target architecture, for example, rolling messages row-wise in a two-dimensional mesh or broadcasting a message to a set of nodes. The communication functions use the process topology functions to direct the messages and the **send** and **receive** primitives in the CON-

LAB language to execute the intended communication by message passing.

3. The third level defines the node process that implements the distributed algorithm. Typically, a node program starts by receiving problem data from the host (see the next level) before entering a loop comprising computations and internode communications. Finally, the results are delivered to the host process. Computations are expressed in the high level constructs and operations offered by the CONLAB language. Communications are expressed by using the communication functions.
4. The fourth level defines the host process that starts the node processes by assigning them to virtual processors. Typically, the host process distributes problem data to the node processes and collects results at the end of the execution. Also the host-to-node communication (and vice versa) is expressed in terms of the communication functions.
5. The fifth and topmost level is the simulation function that defines the interface to the CONLAB user, assigns the host process to a virtual processor, and starts the simulation. Typically, the simulation function communicates problem and architecture parameters (e.g., problem size, input data, and the number of processors).

Conceptually, the described development methodology is well known and hopefully used by most algorithm designers for DMM architectures. The novelty is that CONLAB gives a high-level support for expressing the computations and defining topology and communication functions. CONLAB is extensible in the sense that functions and processes once defined will afterwards exist within the environment and can be used similarly to predefined functions.

The host and node processes define the distributed algorithm, which is a composition of communicating processes. The concept of a host process and node processes is motivated by the commercially available DMM architectures, and in CONLAB, they are considered equal. Within each process all computations are sequential and performed within a single address space (the local memory of the node).

The simulation function can be seen as the problem (or application) level of the abstraction. By changing problem sizes and/or the number of processors, the user can evaluate the parallel performance of the distributed algorithm for the simulated (scalable) DMM architecture.

See also the XYZ abstraction levels defined by Snyder [11, 12] where the X-level is formed by the sequential operations performed within a process. Phases, the parallel composition of processes, define the Y-level. Finally, problems, the composition of phases, define the Z-level.

4 TWO EXAMPLES OF ALGORITHM DEVELOPMENT FOR DMM ARCHITECTURES

In this section we illustrate the algorithm development methodology for DMM architectures, described in the previous section, with two examples. The first example shows how a distributed algorithm for block matrix multiplication on a torus connected two-dimensional mesh of processors is developed and implemented in CONLAB. The second example shows how an already existing serial algorithm for a block QR factorization, expressed as a MATLAB function,* can be parallelized into a ring-oriented distributed block algorithm in CONLAB.

4.1 Block Torus Matrix Multiplication

Given $n \times n$ matrices A, B we will compute $C = A \cdot B$ on a two-dimensional mesh of p processors ($p = s \cdot s$) with torus connectivity. To simplify the notation we assume that $n = nb \cdot s$, that is, A, B (and C) have s^2 blocks denoted A_{ij}, B_{ij} (and C_{ij}), respectively, of size $nb \times nb$. Otherwise, the last block column and block row of the matrices will have rectangular blocks.

The serial *ijk*-algorithm for block matrix multiplication (block inner-product) looks as follows.

```
for i = 1:s
 for j = 1:s
  C_ij := 0
  for k = 1:s
   C_ij := C_ij + A_ik * B_kj
  end
 end
end
```

In the distributed algorithm the processors are organized in a two-dimensional mesh where com-

* MATLAB and CONLAB functions have the same syntax so the sample function could also have been developed in CONLAB.

munication takes place between nearest neighbors as well as around the edge in both directions. In order to compute C_{ij} at node (i, j) we need access to block row i of A and block column j of B.

Initially, the matrices A and B are distributed block-wise among the nodes in the mesh so that node (i, j) has the blocks A_{ik} and B_{kj}, where $k = (i + j - 2) \bmod s + 1$. This means that the matrix A is skewed row-wise and B is skewed column-wise. This distribution scheme allows node (i, j) to compute C_{ij} with only nearest-neighbor communication in the two-dimensional mesh. The main steps of the algorithm [13] for node (i, j) look as follows.

```
C_ij := 0
for t = 1:s
  Receive A_ik and B_kj, where k = (i + j + t - 3) mod s + 1
  C_ij := C_ij + A_ik * B_kj
  Send A_ik to nearest neighbor to the west
  Send B_kj to nearest neighbor to the north
end
```

Notice, the first time node (i, j) receives an A-block and a B-block they are delivered by the host, but the following blocks originate from the nodes to the east and south, respectively.

It would also be possible to let node (i, j) initially hold blocks A_{ij} and B_{ij}. However, this would impose some initial redistribution of the blocks into the skewed pattern described above. In the node algorithm above, we assume that the host process will effect this skewed distribution at once.

Figures 6 through 10 describe the structure of the complete CONLAB program. The topology functions (see Fig. 6) are *coord2node*, *node2coord* which convert coordinates in the two-dimensional mesh into a node number (obtained by the predefined CONLAB function *getpid*) and vice versa, and *north* and *west* which deliver the node number of the neighbors to the north and west, respectively. They use the functions *gray* and *ginv*, which return the gray and inverse gray codes, respectively. The communication function is *roll* (see Fig. 6), which sends a message in a specified direction of the two-dimensional mesh and receives a new message from the opposite direction. In Figures 7 and 8 the CONLAB program for the node process and an accompanying header file *torus.h* are displayed. Figure 9 shows the CONLAB program for the host process. Besides the header file, the host process also includes the file *block.h*, which contains the macros for the automatic matrix blocking facility [6]. Notice, the distribution of A, B to the nodes and the reception of C could also have been defined as communication functions. Finally, the simulation function with the parameters A, B and the size of the two-dimensional mesh is displayed in Figure 10.

4.2 Ring-Oriented Block QR Factorization

In this example we will see how an existing MATLAB program can be parallelized and implemented in CONLAB. We consider a block algorithm to perform a QR factorization of an $m \times n$ matrix A, that is

$$A = Q \cdot R$$

where the $m \times m$ matrix Q is orthogonal and the $m \times n$ matrix R is upper triangular (trapezoidal). The QR factorization can be computed in several ways, for example, by utilizing block Householder transformations [14]. A block Householder transformation Q_i is a product of nb (= the block size) Householder matrices. Here we only consider one block algorithm that is based on the WY representation of block Householder transformations ($Q_i = I + WY^T$) [14]. The main steps of the algorithm are summarized below.

```
Q = I, the identity matrix of order m
nbl := n/nb, number of column blocks (with nb columns) in A
for i = 1:nbl
  Find W and Y such that Q_i = I + WY^T zeroes block column
  i of A below the diagonal (columns [i - 1] · nb + 1:i · nb).
  Apply Q_i to the remaining column blocks of A
  Q := Q · Q_i
end
```

```
% Convert coordinates in the mesh into a node number
function node = coord2node(row, col, side)
        node = gray(row - 1) * side + gray(col - 1);
end

% Convert a node number into mesh coordinates
function [row, col] = node2coord(node, side)
        row = ginv(floor(node / side)) + 1;
        col = ginv(node - floor(node / side) * side) + 1;
end

% Return node number of neighbor to the north of me
function node = north(me, side)
        row = floor(me / side);
        col = me - row * side;
        if (row == 0)
                node = gray(ginv(row) + side - 1) * side + col;
        else
                node = gray(ginv(row) - 1) * side + col;
        end
end

% Return node number of neighbor to the west of me
function node = west(me, side)
        row = floor(me / side);
        col = me - row * side;
        if (col == 0)
                node = row * side + gray(ginv(col) + side - 1);
        else
                node = row * side + gray(ginv(col) - 1);
        end
end

% Send a message M to the node in direction and receive
% a new M from the node in opposite direction
function M = roll(M, direction, type)
        send(M, direction, type, async);
        M = receive(default, type);
end
```

FIGURE 6 Topology and communication functions for block torus matrix multiplication.

The MATLAB function *qrb* for this algorithm is shown in Figure 11.

It is easy to see that the application of Q_i to any one of the remaining column blocks of A can be done independently and the data flow at the block level goes from the left (current block column) to the right (remaining block columns) in a linear list. If we partition data so that each node in the list gets one column block of A then the update can be done in parallel, that is, each node updates its own column block of A with Q_i. Our first attempt to a distributed algorithm goes as follows: node 1 computes Q_1, sends Q_1 to node 2, which sends it to node 3, and so on. Node 2 through p then update their column blocks independently and in parallel. Next, the procedure is repeated but we start with node 2 instead of node 1, and so on until all nodes have computed their part of Q and R. The obvious drawback with this algorithm is that after the first turn node 1 is idle and so on. The remedy is to let each node have more than one column block and wrap-map them so that node i holds column blocks i, $i + p$, $i + 2p$ and so on where p is the number of processors. This gives us a ring topology with a much better load balancing of the computational work. Now, the main steps of the node algorithm looks as follows.

```
#include "torus.h"

% Node process for block matrix multiplication on a
% 2-dimensional mesh with torus connectivity
process torus_node(side)
        % Find out who I am
        me = getpid;

        % Find out my nearest neighbors to the north and to the west
        NORTH = north(me, side);
        WEST = west(me, side);

        % Receive from the host my matrix blocks of A and B
        A = receive(default, AINIT);
        B = receive(default, BINIT);

        % Start timing
        time = timer;

        % Initialize C
        C = zeros(length(A));

        % Loop through all nodes in my block row
        for t=1:side

                C = C + A * B;

                if (t ~= side)
                        A = roll(A, WEST, ABLOCK);
                        B = roll(B, NORTH, BBLOCK);
                end
        end

        % End timing
        time = timer - time;

        % Send the computed C-block to host
        send(C, HOST, CBLOCK, async);

        % Send elapsed time for me to host
        send(time, HOST, TIME, async);
end
```

FIGURE 7 Node program for block torus matrix multiplication.

```
lbl = 0, counter for the number of blocks my node holds
nexttogenerate = 1, keeps track of which process generates the next transformation
for i = 1 : nbl
  if I am nexttogenerate
    1. lbl = lbl + 1
    2. Compute WY factors for my column block lbl
    3. Send W and Y to my right neighbor
  else
    4. Receive WY factors from my left neighbor and send it further
  end
    5. Update my remaining column blocks of A with the new W and Y
    6. Update my row blocks of Q with the new W and Y
end
```

```
% Process id's
% Host id is used together with ParaGraph
% to distinguish it from the nodes
#define HOST        -32768

% Message types
#define NDIM        1
#define AINIT       2
#define BINIT       4
#define ABLOCK      8
#define BBLOCK      16
#define CBLOCK      32
#define TIME        64
```

FIGURE 8 Header file, *torus.h*, for block torus matrix multiplication.

What new functions do we need to realize this algorithm? In step 2 we need a function that computes W and Y, but this function already exists in the serial algorithm (*hshbg* for Householder block generator).

In steps 3 and 4 we need to communicate with the left and right neighbors in the ring. One simple way of doing this is to number the nodes from 0 to $p - 1$ and then the left and right neighbors have the node numbers $me - 1$ and $me + 1$, respectively. The topology functions *left* and *right* are shown in Figure 12. Communication functions *distb* and *recb* for distributing and receiving block wrap-mapped data are shown in Figure 13. For example, the function *recb* receives column (or row) blocks of a matrix distributed with column (or row) block wrap-mapping and packs them into a local array.

Steps 5 and 6 are basically the same as in the serial algorithm. Now, we only apply the WY factors to the blocks of A and Q that each node holds. The CONLAB program for the node process (comprising two parts) and accompanying header file are shown in Figures 14–16. By block wrap-mapping Q row-wise the update of Q is similar to the serial algorithm (see Figs. 11 and 15). For receiving block wrap-mapped data from the host we use the communication function *recb* (see Fig. 13). The host program is straightforward: it assigns the node program to the different nodes, starts the simulation, distributes A and Q, and finally, receives the results from the nodes. The distribution of A and Q is effected by using the communication function *distb* (see Fig. 13). Figure 17 shows the CONLAB code for the host process. Finally, the simulation function *qrb_sim* with parameters A, the block size of A and the size of the ring is displayed in Figure 18.

5 SIMULATION RESULTS FOR AND REAL EXPERIMENTS ON THE INTEL iPSC/2

The two algorithms presented in section 4 have also been implemented in C (block torus matrix multiplication) and Fortran (ring-oriented block QR factorization) and tested on our Intel iPSC/2 hypercube. The machine has 64 scalar SX nodes, each equipped with a 16 MHz Intel 80386 processor with 4 Mbyte memory and a Weitek 1167 (SX) with a theoretical peak performance just under 0.6 Mflops in double precision real arithmetic. The iPSC/2 communication module uses a direct-connect routing module (DCM) on each node, with a bandwidth of 2.8 Mbytes/sec in each direction. Detailed communication benchmarks and models defining the communication time function in CONLAB for the iPSC/2 are presented by Jacobson [8] (see Fig. 3).

Table 1 shows the real and simulated results for the block torus matrix multiplication. Table 2 shows similar results for the ring-oriented block QR factorization. In Tables 1 and 2 we use the following notation: $n (= m)$ the matrix size, p the number of processors, T_p the parallel time in *secs* for p processors, S_p the parallel speedup computed as T_p/T_1, E_p the parallel efficiency computed as S_p/p.

The results for the ring-oriented block QR factorization agree almost completely, whereas the results for the torus matrix multiplication do not. The main reason is the super-linear speedup of the block torus algorithm on the iPSC/2, which is hard to understand completely (the node processors are utilized more efficiently for smaller matrices). CONLAB underestimates the execution time on one processor, resulting in too pessimistic speedup factors for 4, 16, and 64 processors. However, the results from CONLAB correspond to what is expected in theory: for fixed number of processors the parallel efficiency increases as a function of the problem size but keeps below 1.0. The time model makes it impossible to obtain super-linear speedup in CONLAB.

Figures 19 and 20 show window dumps from ParaGraph displaying characteristics of the executions of the ring-oriented block QR algorithm on the iPSC/2 and in CONLAB. The Spacetime diagrams show the communication paths, where a line between two nodes indicates communication, a horizontal line indicates computation, and the absence of a horizontal line means that the processor is idle. Notice that the implementations are not identical. On the iPSC/2 the ring is ordered with Gray code numbering, whereas the ring in the

```
#include <block.h>
#include "torus.h"

process torus_host(p, A, B)
        side = sqrt(p);   % Length of side of two-dimensional mesh
        nb = length(A) / side; % Blocksize of A, B and C

        % Create blocked versions of matrices
        % A, B and C called A_blk, B_blk and C_blk
        blockmatrix(A, A_blk, nb, nb);
        blockmatrix(B, B_blk, nb, nb);
        newmatrix(C, C_blk, rA, cA, nb, nb); %Zero matrices

        % Load node programs and assign virtual processes
        assign(torus_node(side), 0:p-1);

        % Distribute A and B skew-wise to the nodes
        for i = 1:side
                for j = 1:side
                        k = mod(i + j - 2, side) + 1;
                        node = coord2node(i, j, side);
                        send(A_blk(i, k), node, AINIT, async);
                        send(B_blk(k, j), node, BINIT, async);
                end
        end

        % Receive C-blocks from the nodes
        for i = 1:p
                % Receive block of C from any node
                [Block, node] = receive(default, CBLOCK);
                [row, col] = node2coord(node, side);
                C_blk(row, col) = Block;
        end

        % Check the result by computing the 2-norm of the residual
        disp('norm(A*B-C) =', norm(A * B - C));

        % Receive elapsed times from the nodes
        time = [];
        for i=1:p
                time = [time, receive(default, TIME)];
        end
        % Parallel time is the maximum over the node elapsed times
        disp('Parallel time =', round(max(time) / 1000), ' msec');
end
```

FIGURE 9 Host program for block torus matrix multiplication.

```
function dummy = torus(A, B, side)
        % Compute number of nodes
        p = 2^side;

        % Start host process on virtual processor p
        % giving it process id HOST
        assign(torus_host(p, A, B), p, HOST);
        simulate
end
```

FIGURE 10 Simulation function for block torus matrix multiplication.

```
function [A, Q] = qrb(A, nb)
% QRB computes a QR-factorization of A by utilizing block-Householder
% transformations, where nb is the column block size.
% This variant uses the WY-representation (method 1, Bischof-Van Loan).
        [ma, na] = size(A);
        nbl = fix((min(ma,na) + nb-1))/nb; %number of column blocks
        Q = eye(ma);
        for k = 1 : nbl
                s = (k-1)*nb + 1;          % start of current block
                e = min(s+nb-1, min(ma,na)); % end of current block
                if k == nbl % last column block may have different size
                        nb = e-s+1;
                end
%               Generate the block Householder transformation that
%               QR-factorizes A(s:ma,s:e)
                [Wk, Yk] = hshbg1(A(s:ma,s:e));

%               Apply the block Householder transformation to remaining blocks
                A(s:ma,s:na) =  A(s:ma,s:na)+Yk*(Wk'*A(s:ma,s:na));

%               Accumulate block Householder transformation
                Q(:, s:ma) = Q(:, s:ma) + (Q(:, s:ma) * Wk) * Yk';
        end
end
```

FIGURE 11 MATLAB function for block *QR* factorization.

CONLAB implementation is numbered consecutively. Because the algorithm only uses nearest-neighbor communication, the time model for multi-hop communication is never involved and the node numbering does not affect the results. Similar diagrams for the block torus matrix multiplication are shown in Figures 21 and 22. In this example we have a more complex communication pattern, but still the communication paths of the CONLAB simulation are accurate. The overhead from the tracing facility on the iPSC/2 is more visible in Figure 21 than in Figure 19. Both these examples do not use a communication channel of the iPSC/2 for more than one message. This means that we will not encounter any communication channel contention. At present, contention in the communication system is not modeled in CONLAB. Therefore, the results from a simulation in CONLAB can be too optimistic for algorithms that involve contention in the real communication system. However, our experiences show that we still get a fair picture of the relative parallel performance when changing the size of the problem or the size of the scalable DMM architecture.

```
function res = left(me, p)
% Who is my left neighbor in a ring of p processors?
% Processors numbered 0:p-1
    if (0 <= me) & (me <= p-1)
                res = me - 1;
                if (res == -1), res = p-1; end
        end
end

function res = right(me, p)
% Who is my right neighbor in a ring of p processors?
    if (0 <= me) & (me <= p-1)
                res = me + 1;
                if (res == p), res = 0; end
        end
end
```

FIGURE 12 Topology functions for a ring of processors.

6 CONCLUSIONS AND FUTURE DEVELOPMENT

Algorithm development for DMM architectures by using the CONLAB environment has been illustrated by two examples concerning matrix computations. The algorithm design process follows a development methodology for DMM algorithms that is based on different levels of abstraction of

```
function distb(A, b, nb, p, wise)
% Distribute matrix A with block row/column wrap-mapping
% on a ring of processors.
% b = block size, nb = # blocks, p = # processors
% wise = distribution (ROW or COL)
#include    "qrb.h"
        [m n] = size(A);
        node = 0;
        for i = 0:nb-1
                s = i*b + 1;     e = min(s+b-1,n);
                if wise == COL
                        send(A(:,s:e),node,MDIST, async);
                else    % wise == ROW
                        send(A(s:e,:),node,MDIST, async);
                end
                node = right(node, p);
        end
end

function [A, mynbl] = recb(nbl, me, p, wise, type)
% Receive in A my blocks of a matrix distributed
% by block row or block column wrap-mapping.
% nbl = total number of row/column blocks distributed
% p = number of processors
% wise = ROW if row-wise or COL if column-wise distribution
% mynbl = my number of blocks
#include        "qrb.h"
        A=[];
        if me == HOST
                mynbl = nbl;
                from = 0;
        else
                mynbl = fix(nbl / p);
                if me < nbl - mynbl * p
                        mynbl = mynbl + 1;
                end
                from = HOST;
        end
        for i = 1:mynbl
                if wise == COL
                        A = [A, receive(from,type)];
                else    % wise == ROW
                        A = [A; receive(from,type)];
                end
                if me == HOST
                        from = right(from,p);
                end
        end
end
```

FIGURE 13 Communication functions for distributing and receiving block wrap-mapped data.

the problem, the target architecture, and the CONLAB language itself. Different tools and ways for debugging algorithms and interpreting and evaluating the results have also been discussed.

CONLAB has extensively been used in parallel computing courses at our University. Further, the parallel algorithms presented [1, 2, 15] have first been designed in CONLAB and then explicitly translated by hand to Fortran or C. By designing and experimentally verifying the parallel algo-

```
process qrb_node (w, nbl, nbla, nblq, m, p)
% Node process for ring topology block-Householder QR factorization
% w = column block size, nbl = # block Householder transformations,
% nbla = # column blocks of global A, nblq = # column blocks of global Q
#include        "qrb.h"
        me = getpid;
%       Receive my column blocks of A, store in A
        [A, nablk] = recb(nbla, me, p, COL, MDIST);
        [ma, na ] = size(A);
%       Receive my row blocks of Q, store in Q
        [Q, nqblk] = recb(nblq, me, p, ROW, MDIST);
        [mq, nq ] = size(Q);
        lbl = 0; % counter for WY-factors generated by me
        nexttogen = 0; % keeps track of which process generates next WY-factors
        LEFT  = left(me, p);     % my neighbors
        RIGHT = right(me, p);
        time = timer; flps = flops;          % time this process
        for i = 1: nbl
                if nexttogen == me          % my turn to generate next WY-factors
                        lbl = lbl + 1;
                        sl = (lbl - 1) * w + 1; s = (i-1)*w+1;
                        el = min(lbl*w, min(na, sl + m - s));
%                       Generate WY-factors that factorizes A(s:m, sl:el)
                        [Wk, Yk] = hshbg(A(s:m, sl:el));
                        send([Wk, Yk], RIGHT, WYFACTOR, async);
                        send([s, el-sl+1], RIGHT, IND, async);
                else
                        WY = receive(LEFT, WYFACTOR);
                        se = receive(LEFT, IND);
                        if RIGHT ~= nexttogen %WY not generated by RIGHT
                                send(WY, RIGHT, WYFACTOR, async);
                                send(se, RIGHT, IND, async);
                        end
                        s = se(1); wl = se(2);
                        Wk = WY(:, 1:wl);
                        Yk = WY(:, wl+1:2*wl);
                end
%               Apply WY-factors to my remaining blocks
                sl = lbl* w + 1; if nexttogen == me, sl = sl - w; end
                if sl <= na
                        A(s:m,sl:na)=A(s:m,sl:na)+Yk*(Wk'*A(s:m,sl:na));
                end
```

FIGURE 14 Node program for ring-oriented block *QR* factorization, part 1.

Table 1. Real and Simulated Performance Results of Torus Matrix Multiplication

Torus $C = A \cdot B$		*iPSC/2*			*CONLAB*		
n	p	T_p	S_p	E_p	T_p	S_p	E_p
128	1	10.674	1.0	1.00	8.42	1.0	1.00
	4	2.29	4.7	1.16	2.21	3.8	0.95
	16	0.63	17.1	1.07	0.65	12.9	0.80
	64	0.19	55.6	0.87	0.30	28.2	0.44
256	1	88.07	1.0	1.00	67.24	1.0	1.00
	4	21.58	4.1	1.02	17.22	3.9	0.98
	16	4.66	18.9	1.18	4.67	14.4	0.90
	64	1.27	69.5	1.09	1.54	43.7	0.68
384	1	—	—	—	226.79	1.0	1.00
	4	72.54	1.0	1.00	57.62	3.9	0.98
	16	15.99	4.5	1.13	15.20	14.9	0.93
	64	4.08	17.8	1.11	4.55	49.8	0.78

```
% Accumulate block Householder transformations
                if mq ~= 0
                        Q(:,s:nq) = Q(:,s:nq) + (Q(:,s:nq)*Wk) * Yk';
                end
                nexttogen = right(nexttogen,p); %next process to generate WY factor
        end
        total = timer - time;
        arith = (flops - flps) * floptime;
        com = total - arith;
        disp('Time pid = ', me, '  Total = ', total, '  Arith = ', ...
        arith, '  Com = ', com)
%       Results to the host
        for i = 1: nablk
                send(A(:, (i-1)*w+1:min(i*w, na)), HOST, ARES, async);
        end
        for i = 1: nqblk
                send(Q((i-1)*w+1:min(i*w, mq), :), HOST, QRES, async);
        end
end % process bqr_node
```

FIGURE 15 Node program for ring-oriented block *QR* factorization, part 2.

```
% Host id is used together with ParaGraph
% to distinguish it from the nodes
#define HOST            -32768

% Message types
#define MDIST           20
#define WYFACTOR        30
#define IND             40
#define ARES            50
#define QRES            60

% Type of matrix
#define ROW             100
#define COL             110
```

FIGURE 16 Header file, *ring.h*, for ring-oriented block *QR* factorization.

```
process qrb_host(A, nb, p)
% Host for ring-oriented block-Householder QR factorization
% nb = blocksize, p = number of processors
#include        "qrb.h"
[m,n] = size(A);                        % get size of matrix
nbl = fix(((min(m,n)+nb-1))/nb);        % number of Householder transformations
nbld = fix((n + nb - 1)/nb);            % total number of column blocks
nblq = fix((m + nb - 1) /nb);           % number of q blocks
Q = eye(m);
assign(qrb_node(nb,nbl,nbld,nblq,m,p), 0:p-1);

% Distribute A with block-column wrap mapping
distb(A, nb, nbld, p, COL);

% Distribute Q with block-row wrap mapping
distb(Q, nb, nblq, p, ROW);

% Receive results
[R x] = recb(nbld, HOST, p, COL, ARES);
[Q x] = recb(nblq, HOST, p, ROW, QRES);
end
```

FIGURE 17 Host program for ring-oriented block *QR* factorization.

Table 2. Real and Simulated Performance Results of Ring-Oriented Block *QR* Factorization

Ring *QR*		*iPSC/2*			*CONLAB*		
$m = n$	p	T_p	S_p	E_p	T_p	S_p	E_p
64	1	5.406	1.0	1.00	5.462	1.0	1.00
	4	1.58	3.4	0.86	1.71	3.5	0.87
	8	0.95	5.7	0.71	0.93	5.9	0.74
	16	0.64	8.5	0.53	0.60	9.1	0.57
	32	0.49	11.0	0.34	0.42	12.5	0.39
128	1	43.23	1.0	1.00	42.23	1.0	1.00
	4	11.26	3.8	0.94	11.28	3.8	0.94
	8	6.22	6.8	0.85	6.14	6.9	0.86
	16	3.69	11.4	0.71	3.56	11.9	0.74
	32	2.44	17.3	0.54	2.26	18.7	0.58
	64	1.82	23.3	0.36	1.59	26.6	0.42
240	1	274.70	1.0	1.00	274.40	1.0	1.00
	4	70.92	3.9	0.97	70.89	3.9	0.97
	8	37.20	7.4	0.92	37.11	7.4	0.92
	16	20.73	13.3	0.83	20.28	13.5	0.85
	32	12.43	22.1	0.69	11.87	23.1	0.72
	64	8.28	33.2	0.52	7.61	36.1	0.56

```
function dummy = qrb_sim(A, blksize, p)
%       Simulation function for a ring-oriented
%       block Householder QR factorization.
%       A = matrix to be factorized, blksize = block size,
%       p = number of processors
%       Uses processes qrb_host and qrb_node
#include        "qrb.h"

%       Start the host process on virtual processor p
%       giving it process id HOST
        assign(qrb_host(A, blksize, p), p, HOST);
        simulate
end
```

FIGURE 18 Simulation function for ring-oriented block *QR* factorization.

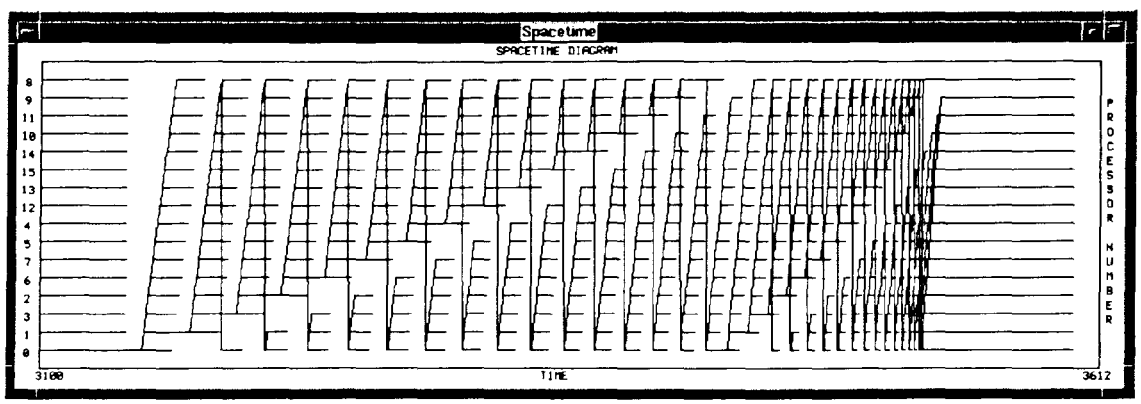

FIGURE 19 Spacetime diagram for ring-oriented *QR* factorization on the Intel iPSC/2 hypercube.

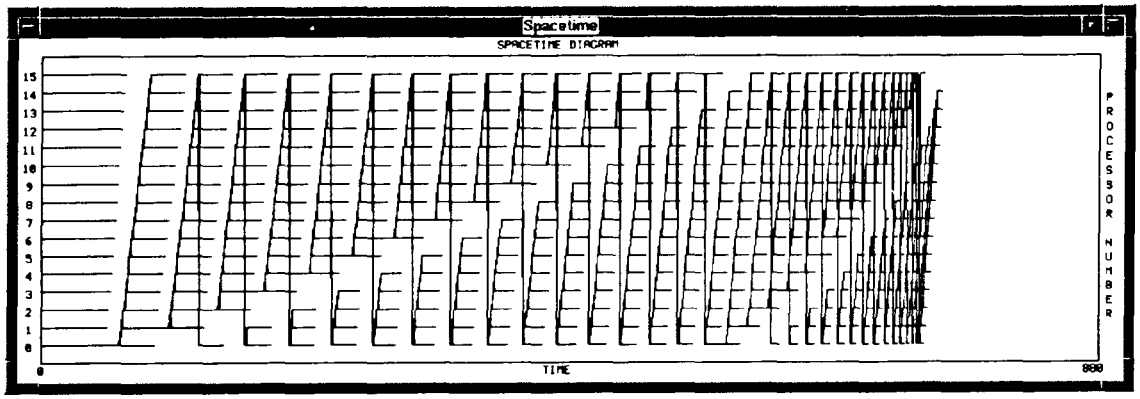

FIGURE 20 Spacetime diagram for ring-oriented QR factorization simulating the iPSC/2 in CONLAB.

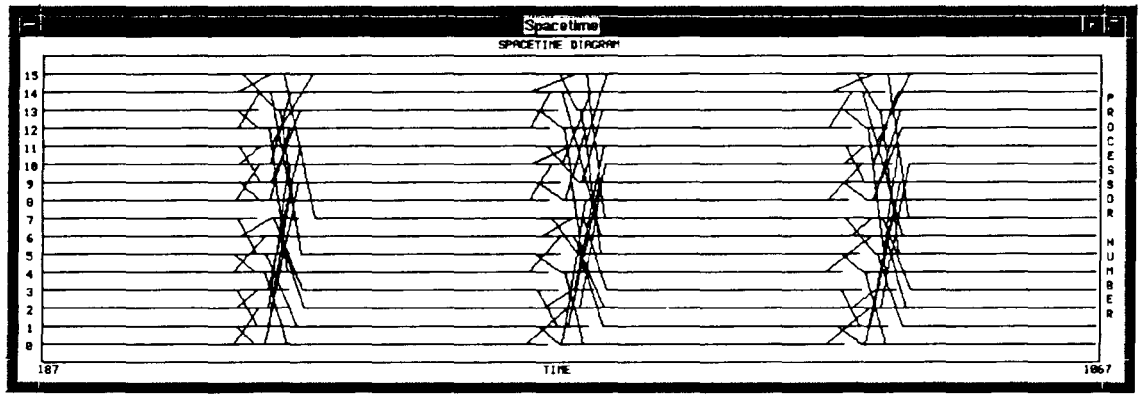

FIGURE 21 Spacetime diagram for block torus matrix multiplication on the Intel iPSC/2 hypercube.

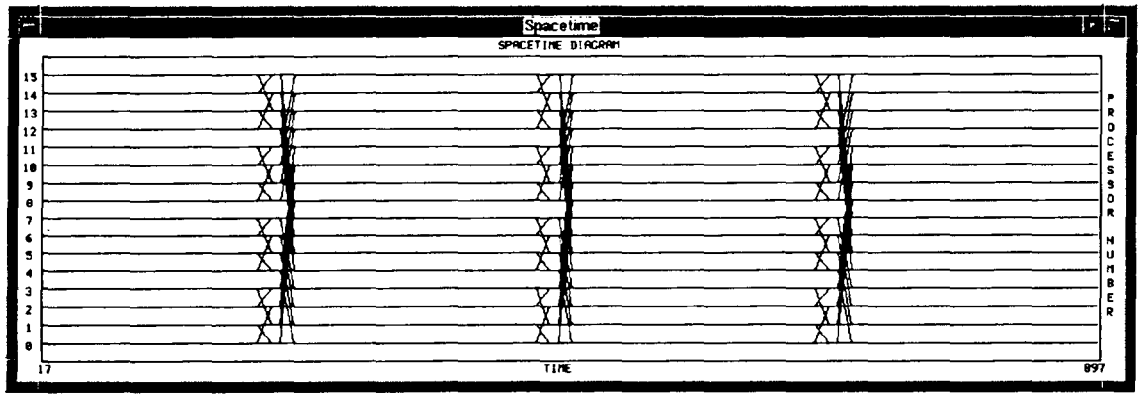

FIGURE 22 Spacetime diagram for the block torus matrix multiplication simulating the iPSC/2 in CONLAB.

rithms in CONLAB the efficiency and the quality of the development process have been improved appreciably.

The future directions of development of CONLAB are twofold. First, the introduction of time models for several DMM architectures is under way. A mechanism for choosing a DMM architecture and corresponding architecture parameters exists already. Second, and in parallel, the development of a CONLAB compiler (translator) has started. It takes a DMM algorithm in CONLAB syntax as input and will produce a similar C program to run on the target architecture. As in CONLAB, the numerical computations will be based on LAPACK [3]. Further, the communication will be based on efficient and portable implementations of topology and communication functions (see sections 3 and 4).

ACKNOWLEDGMENTS

We would like to thank Cleve Moler and associates at The Mathworks, Inc. for creating MATLAB, without whose existence CONLAB would never even have been thought of. Financial support has been received from the Swedish Board of Technical Development (NUTEK) under contract STU-89-02578P.

REFERENCES

[1] K. Dackland, E. Elmroth, B. Kågström, and C. Van Loan, "Parallel block matrix factorizations on the shared-memory multiprocessor IBM 3090 VF/600 J," *Int. J. Supercomput. Appl.*, vol. 6, pp. 69–97, 1992.

[2] K. Dackland and E. Elmroth, "Parallel block matrix factorizations for distributed memory multicomputers," Technical Report UMINF-92.03, Institute of Information Processing University of Umeå, S-901 87 Umeå, Sweden, 1992.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide*, Philadelphia, PA: Society for Industrial and Applied Mathematics, SIAM Publications, 1992.

[4] C. Moler, J. Little, and S. Bangert, *PRO-MATLAB User's Guide*, The Mathworks Inc., 1987.

[5] J. Eriksson, P. Jacobson, B. Kågström, and E. Lindström, *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, Philadelphia, PA: SIAM Publications, 1990, pp. 406–412.

[6] P. Jacobson, "The CONLAB environment," Technical Report UMINF-173.90, Institute of Information Processing, University of Umeå, S-901 87 Umeå, Sweden, 1990.

[7] P. Jacobson, "The CONLAB simulator—execution and scheduling of parallel computations," Technical Report UMINF-92.08, Institute of Information Processing, University of Umeå, S-901 87 Umeå, Sweden, May 1992.

[8] P. Jacobson, "Detailed communication benchmarks of Intel iPSC/2 and iPSC/860 hypercubes," Technical Report UMINF-92.05, Institute of Information Processing, University of Umeå, S-901 87 Umeå, Sweden, May 1992.

[9] M. Heath and J. Etheridge, "Visualizing the performance of parallel programs," IEEE Software, vol. 8, pp. 29–39, 1991.

[10] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley, "PICL: A portable instrumented communication library," Technical Report ORNL/TM-11130, Oak Ridge National Laboratory, 1990.

[11] L. Snyder, *Language and Compilers for Parallel Computing*, Boston, MA: MIT Press, 1990, pp. 470–489.

[12] L. Snyder, "Applications of the 'Phase Abstractions' for portable and scalable programming," Technical Report, Department of Computer Science and Engineering, University of Washington, FR-35, Seattle, Washington 98195, USA, 1990.

[13] V. Cherkassky and R. Smith, "Efficient mapping and implementation of matrix algorithms on a hypercube," *J. Supercomput.*, vol. 2, pp. 7–27, 1988.

[14] G. Golub and C. Van Loan, *Matrix Computations*, second edition, Baltimore, MD: John Hopkins University Press, 1989, pp. 213–214.

[15] B. Kågström and P. Poromaa, "Distributed and shared memory block algorithms for the triangular Sylvester equation with Sep^{-1} estimators," *SIAM J. Matrix. Anal. Appl.*, vol. 13, pp. 90–101, 1992.