A Small Parallel C Simulator

Manuel Mollar
Depto. de Informática
Universidad Jaume I
Aptdo. 242, 12071 Castellón, Spain

Abstract

The C language is an interesting choice for parallel programming with the transputer. We present a module built in C that allows a program written in INMOS Parallel C to be run in any computer with minor changes, using a sequential compiler. Only the language is simulated, not the architecture, providing fast execution and reasonably accurate timing.

1 Introduction.

Our simulator allows the compilation and execution of parallel programs written in Parallel C using our favorite programming toolset in the corresponding computer. The main goal is the *simulation of the language*, not the architecture. In consequence, the debugging facilities will be those of the programming tool, so the training effort is minimum. In short, the main advantages of the built simulator are four: debugging facilities, high execution speed, deadlock detection and the possibility of running without the transputer hardware. It has been written in ANSI C to guarantee portability. After having developed it using the Borland C IE on a PC-clone, it has been compiled under HP UX without any change.

In §2 we describe the simulation guidelines and give some definitions, in §3 and §4, the necessary changes to program source are described. In §5 and §6 the basic simulation strategy and algorithms are given, showing the debugging possibilities in §8 and §9. In §7 we explain the available multiprogramming features.

2 The simulation.

The simulator is constructed as two include files and one module that must be linked with the parallel program. It provides the routines for communication, and a transparent concurrence mechanism to simulate the parallelism. The chosen concurrence strategy is based in a co-routine scheme built using the *setjmp* C package. Of course, a round-robin scheduler would be more realistic, but the difficulties and the loss of portability are evident. Furthermore, most debuggers will not run with this scheme. Co-routine based concurrent execution is far from the behavior of the parallel program. However, once the transfer strategy has been understood, the sequential trace of the program is very convenient for a good tracking of the parallel algorithm.

For every processor, the simulator distinguishes (and can compute) among *Total Time (TT), Input Waiting Time (IWT), Output Waiting Time (OWT), Communication Time (CMT),* and *Computation Time (CPT).* Last one is the employed by the simulating processor (in all that follows, the host processor) in executing any no-communication instruction. A scale factor is computed to return the real transputer times (T805).

The communication costs (CMT) are estimated with a usual theoretical model: estimate the cost of the link set up and the cost of the transmission of one byte. The total cost is: cost of set up + cost per byte * no. of bytes. Both can be manual or automatically adjusted. Setting them to 0,0, the simulator will show the degree of parallelism of the algorithm. The communication costs are only accounted in the processor that does the output operation (in all that follows, the output processor); in the input processor, it is accounted as Input Waiting Time, defined as the time from the input operation request to its completion. By Output Waiting Time we designate the expired one from the output request till the input operation beginning.

Always, we know the Total Time per processor, so the maximum one is the time of the parallel algorithm. We also know the total computation time, as the addition of the Computation Time of every processor. Then, the simulator can compute the *speed up* of the algorithm for any given number of processors respect to the same algorithm executed in one processor.

In the simulation, a processor executes its code sequentially until it requires to communicate with another. Depending on the situation of the other, execution is transferred to it. The transfer routine accounts the expended CPU time. The algorithmic effort is centered in the communication routines, as we will explain.

The number of physical links in the transputer can be extended in the simulation to any number. So, an algorithm designed for a n-cube that runs directly in the T805 transputer for n = 1,2,3, can be tested for any value of n, to prove its correction, simply increasing the number of links at configuration time.

3 The tailored program.

In the transputer system, configured executable code may be constituted by different programs for the different processors [1]. Now, by evident reasons, only a program can be present. The solution is that *each simulated processor executes a routine of the tailored program.* This routine does not need to be different for every processor, the configuration mechanism allows to allocate the same routine for several processors.

Thus, the first tailoring operation is to join the original programs into a single one, maintaining the multiple file structure, if desired. To do it, all the *main* routines must be renamed. Then, a new *main* routine must be created, intended to configure the network and to start the simulation. To avoid the construction of a parser for the files accepted by *icconf* [1], the simulator includes a set of routines that implements the configuration task.

The joining process has a drawback concerning to global variables. Suppose that in the transputer network configuration, a program must be shared by several processors. In the tailored code, we want to allocate the same routine for these processors. Then, all the global variables from the original program must be converted into local ones to the renamed routine. The presence of global data in the tailored program implies the simulation of a shared memory. The only alternative is the replication of the variables, with the corresponding naming task or the inclusion in a vector, but this is not a serious approach.

To avoid having a program for the simulator and others for the transputer, we can use the C preprocessor. Using the #ifdef directive, code can be tailored at compilation time, using the appropriate definitions. For example, suppose that we want to run a program in processor 0 and another in the rest of processors. We can write the code for a single file like this:

```
/* Here the headers for all the processors */
#ifndef procR
/* Here special headers for processor 0 */
#endif
#ifndef procR
#ifdef proc0
  int main()
#else
  void RoutineFor0 ()
#endif
/* Here the code for processor 0 */
#endif /* #ifndef */
#ifndef proc0
#ifdef procR
  int main()
#else
  void RoutineForRest ()
#endif
/* Here the code for the rest of processors */
#endif /* #ifndef */
#ifdef InSimulator
  void main()
  /* Here the code for configuring */
#endif
```

In this example, for the real machine, we must define *proc0* at compiling time to generate code for processor 0, define *procR* to generate code for the rest of processors. When in the simulator, *InSimulator* is already defined, allowing code generation for the simulator. So, the tailoring process becomes automatic. If the multiple file scheme is maintained, a more simple use of the preprocessor will give the correct result. Using the *#ifdef* directive for the *main* and selecting the files at link time, will be enough.

4 The configuration routines.

These procedures are similar to those used by *icconf* to make them easy to use. The main difference comes from the tailored program: each simulated processor holds a routine of the program, not a different program (like in the original occurs). Furthermore, no distinction is made between process and processor, so the original sentences *place* and *use* are joined in a single one, *Place*. It sets which routine is allocated in what processor, with an estimate of the extra workspace that the routine needs (in the stack), as we will explain. Another parameter to *Place* is the result to a call to *Interface*. This routine implements partially the original interface: only I/O channels and integer values are supported as parameters. The process Id is treated as a special case, to economize memory. A call to *Place*, will be, for example:

```
Place(rout, extra, proc,
      Interface(5,iPId,iInpChan,East,iOutChan,East,
                iInpChan,West,iOutChan,West)),
```

which will place a routine *rout* in processor *proc* with the extra stack *extra* and with an interface including five parameters: the PID, an input process (designed by iInpChan) for link East (1), an output process for the same link, and the same for link West.

Channel specification in the interface implies explicit determination of the physical link on which the channel will be placed, so the original *place* sentence applied to channels is not necessary (and not implemented). To connect two processors, the *Connect* procedure is used, specifying the two processors and the two links. Establishing connection between two processors by a determined link implies the existence of two channels for each processor, which can be used by the placed routine if they are specified in the interface construction.

Despite the differences with the original configuration process may seem substantial, this simplified scheme allows in practice setting up most of the configuration schemes, in the desired way: the use of *get_param* by the program does not need to be modified.

Furthermore, dynamic creation of interfaces is possible, allowing their construction at run time. Using this facility, a set of built in topologies is given with the simulator, included in *misc.h*, allowing quickly starting:

- A bi-directional ring of processors,
- A crossbar of any number of processors,
- A full tree with any number of sons by node and any deepth,
- A wrapped mesh with any number of dimensions with any number of processors by dimension, and
- A hypercube of any dimension.

Obviously, when more than four links are needed, the algorithms cannot be directly implemented in the T805 transputer. Here, the interface is constructed using the links in the order 1,2,3,0,4,5, ... So, compatibility is maintained when possible.

A complete description may be found in the include file *misc.h*.

5 Process management.

Each processor is simulated with a co-routine that executes one of the renamed routines. Each processor has its local clock. *Process work space is allocated in the stack*, including the local data. Only three procedures implement process management. The routine *StartSystem* called when the configuration task is done, assigns a PID to itself (we refer to it as the initial process) and transfers to the processor 1. Using an informal notation, the algorithm is simple:

```
set the initial time
transfer(1)
repeat
  for every non finished process do
    transfer to it
until no process
```

The *Transfer* routine *is only called internally by the communication routines*, so context switching is transparent to the user. The algorithm for *Transfer(PID)* follows.

```
localclock+= current time of host processor -
             previous initialtime
if process with this PID does not exist then
  set the work space for the current process
  call the placed routine
  exit_terminate(0)
else if process has finished then error
else initialtime= current time of host processor
    transfer to the process with this PID
```

The other routine is *exit_terminate*, that simply marks the process as terminated and transfers execution to the initial process, entering the previous repeat loop, transferring control to any other non finished process. This loop is also intended to schedule any process that performs some action after the last call to some communication routine, because process scheduling is *only carried out by the communication routines.*

It is usual that the first simulated routine called by any program is *get_param*. This is not necessary in the real machine, but it is in the simulator. The first time that a program calls *get_param*, it does its expected work, and then starts the next process. The declared local variables are already allocated in the stack, so the extra stack is only intended to allow possible calls to other routines, and acts as a work space for the process. The process creation ordering is 1, 2, ..., n-1, 0, so the remaining of the stack is available for proc. 0, despite its extra stack (the specified). This is very convenient, because proc. 0 may need a great amount of stack to perform I/O.

6 The communication routines.

The data structure for every process is an array of channel descriptors. One input channel and another output one are allocated for each physical link, no matter if they were not specified in the interface, for simplicity. Each descriptor holds what link in what processor is connected to it. The communication strategy is a simple buffer operation: the data passed to ChanOut are copied into a buffer pointed by the input channel descriptor of the connected processor. ChanIn will deallocate the buffer after copying it to its destination.

When a process calls ChanIn, data may be not ready. In our concurrence scheme, waiting for the message consists in transferring control to the partner process, from which data are expected. Calling ChanOut by the output process, execution is transferred to the input process. The specification for ChanIn is given in [2] and the basic algorithm may be:

```
if message not received then
  transfer to the output process
/* execution continues here if a transfer to this
                                 process occurs */
if data have been received then
  copy information
  deallocate buffer
else deadlock
```

And for ChanOut:

```
allocate buffer
copy message to it
transfer to the input process
if data not deallocated then
  deadlock
```

However this is not valid. Deadlocks will be detected where they do not occur, due that the control may be transferred to the input process by another process. For example, suppose this situation:

Process 0	Process 1	Process 2
ChanIn from 1	ChanOut to 2	ChanIn from 1
ChanIn from 2	ChanOut to 0	ChanOut to 0

Starting from processor 0, control will return to it from processor 2, and data are not ready in the channel to proc. 1. If control is **another time** transferred to proc. 1, the second ChanOut (to proc. 0) will be performed, and the example communication will terminate.

Deadlock detection is possible establishing the maximum number of transfers (always to the same processor). Observe that anyway, with three processors at most two transfers will be necessary to schedule the system. With n processors, if processor 0 performs a ChanIn to communicate with proc. 1, at most the other n-2 procs. can demand communication with proc. 0 without knowing for certain that deadlock will occur, so at most n-2 extra transfers can be admitted, no matter from which processor come. If the last transfer does not carry the message, deadlock occurs. So the algorithm for ChanIn is this:

```
while (message not received) and
        (number of transfers is less than $n$) do
  transfer to the output process
if data not received then
  deadlock
else ...
```

Observe that during the extra transfers, control pass through processes that can be doing output, so a similar loop must appear in ChanOut.

For each input channel, IWT and the number of input operations (NIO) are accounted. For each output channel, OWT, the number of output operations (NOO) and the number of transferred bytes (NTB) are registered. The values NOO and NTB are accounted in order to give complete information to the user. Communication times are no explicitly accumulated. So, the computation time (CPT) is obtained from:

$$TT = CPT + \sum_{Channel}[IWT + OWT + NOO * (\text{channel set up time}) + NTB * (\text{time per byte})].$$

While TT is measured in the transfer operation, considering the previous algorithms, the communication times must be determined in the ChanIn routine. Transfer operations in ChanIn and ChanOut will cause undesired CPU time spent. So, considering the high performance of the original Parallel C routines, the algorithmic CPU time is discarded and the computation time for ChanIn and ChanOut is reduced to the minimum updating directly TT for the process. Supposing the data received and TI the processor time at which ChanIn begins and TO the (different) processor time at which the corresponding ChanOut begins, the algorithm for time measuring is:

```
if $TI < TO$ then
  /* input process is waiting */
  $wait = TO - TI + CMT$
  $TO+ = CMT$
  $TI = TO$
  $IWT+ = wait$
else
  $wait = TI - TO$
  $TO = TI + CMT$
  $TI = TO$
  $IWT+ = CMT$
  $OWT+ = wait$
```

The routines *ChanInChanFail* and *ChanOutChanFail* described in [2] have not been implemented. However, *ChanInTimeFail* or *ChanOutTimeFail*, are very interesting for asynchronous parallel programming. Now, suppose that *timeI* is the max. waiting time for ChanInTimeFail and *timeO* the same for ChanOutTimeFail, the algorithm for ChanInTimeFail follows.

```
repeat   /* to handle a ChanOutTimeFail that fails */
  repeat perform transfers
  until data received or deadlock condition
  if no data received then
    if $timeI \geq 0$ then
      $TI+ = timeI$
      $IWT+ = timeI$
      return failure
    else
      deadlock
  else
    compute $CMT$
    if $TI \leq TO$ then
      $wait = TO - TI$
      if $(timeI \geq 0)$ and $(timeI < wait)$ then
        /* failing */
        $aux = TI + timeI$
        $wait = timeI$
        $failI$ =true
      else
        $aux = TO + CMT$
        $wait = aux - TI$
        $TO = aux$
      $IWT+ = wait$
      $TI = aux$
    else
      $wait = TI - TO$
      if $(timeO \geq 0)$ and $(timeO < wait)$ then
        $aux = TO + timeO$
        $wait = timeO$
        $failO = true$
        $CMT = 0$
      else
        $aux = TI + CMT$
      $OWT+ = wait$
      $TO = aux$
      $IWT+ = CMT$
      $TI+ = CMT$
  if not $failI$ then
    copy data and deallocate
until $failI$ or not $failO$
return $failI$
```

Observe that the previous algorithm for ChanIn is embedded in this one. So, ChanInTimeFail is the basic input routine: ChanIn is ChanInTimeFail with *timeI = -1*. The same applies for ChanOut and ChanOutTimeFail, where simpler changes are made to return failure when deadlock or not message in *timeO*.

7 Multiprogramming features.

The first requirement for an optimal simulator is that any program written in Parallel C may run on the simulator. However, this is not true here. The simulator does not support multiprogramming in a transputer. So all the calls relative to process management are not supported, except those of time measuring. Really, we have originally designed the simulator to *develop and test* some numerical parallel algorithms. In this context, one process per processor is a usual practice, to obtain greater efficiencies.

However, several multiprogramming features have been implemented, as follows. The fact that the simulator does not support explicit multiprogramming, inhibits the possibility of performing concurrent I/O. Concurrent input, for example, is useful to make simultaneous input from different links. Even in a single link communication, concurrent I/O can be very useful to increase efficiency by means an intermediate message buffering. This possibility is valid for any parallel program. Then, in the simulator, an additional option is available at configuration time to stay that all I/O is handled by an intermediate buffering process for every channel. The code executed by this virtual process follows.

For input operations:

```
repeat forever
  read message into auxiliary buffer (at time $Ti$)
  send it to the user process (at time $TI$)
```

For output operations:

```
repeat forever
  read message from user process into auxiliary
          buffer (at time $TO$)
  send it to the partner (at time $To$)
```

In the real machine, the communication times may be strongly improved, but additional memory will be needed. In the simulation, no process is created, and no memory is needed for buffers. Simply, ChanIn will compute times in a different way. Although in the transputer process contention will disappear in most cases (using the buffering), the simulated behavior is the same that without the buffering, by obvious reasons.

The suggested use of this option follows: when the program works, activate the option and then, compare execution times; if they are really improved, building the buffering with auxiliary processes in the transputer may be justified.

In the simulator, concurrent mode is not compatible with time fail routines. Considering than $Ti \leq TI$ and $TO \leq To$, three situations are possible: $Ti \leq TI < To$, $To < Ti \leq TI$ and $Ti \leq To \leq TI$. The algorithm for ChanIn follows:

```
if $TI < To$ then
    $IWT+ = To - TI + TC$
    $To+ = TC$
    $TI = To$
else
    if $To < Ti$ then
        $To = Ti + TC$
        if $To > TI$ then
            $IWT+ = To - TI$
            $TI = To$
    else
        $To+ = TC$
        if $To > TI$ then
            $IWT+ = To - TI$
            $TI = To$
$Ti = TI$
```

The WT is incremented in ChanOut, that, before transfering does:

```
if $To$   /* previous, actualized by ChanIn */
        $> TO$   /* actual */ then
    $OWT+ = To - TO$
    $TO = To$
else
    $To = TO$ /* for the next */
```

8 Debugging facilities.

The simulator has a trace mode in which several events are logged to the output. The events are mainly the relatives to communication activity, including waiting times. Combining the trace with the possibility of using *printf* instructions from any processor, a classical debugging method is obtained.

Of course, we prefer the possibilities given by interactive symbolic debuggers. The use of the step by step execution founds here a similar situation to those that appears in event-driven programs: a step may result in an excessive number of operations for debugging purpose. Suppose that a call to *ChanOut* is executed. It will perform a transfer to the input process, that in its turn can cause another transfers. So, when control returns to the next instruction, a lot of work may be done, and if we search for some error, it may appear in this step.

Breakpoints are the recommended tool. If we put them in the critical points of the program, each time that the control passes through the breakpoint, execution will stop. If several processors share a same routine (it is usual), the breakpoint will affect to the first processor that arrives to it. When conditional breakpoints are available, the processor selection is allowed using the variable *Process*, which returns the current process. Evidently, this variable must not be used into the program.

For debugging purposes, synchronization is an important task. In the transputer, under *idebug*, putting breakpoints in the different processors allows pausing all the processors at the desired point. Without the debugger, synchronization usually needs to be done in the transputer by means of message passing. In the simulator, only one process will arrive at his breakpoint at a time, due the sequential nature of the simulation; so another way of synchronization is necessary. The routine *Synchronize*, located in the desired points, simulates such a message passing, without any channel use. Processes are blocked inside *Synchronize* until every one calls it. Then execution continues in processor 0. But *Synchronize*, in contrast with message passing, does not affect CPU time; it can appear in the source code as it is needed, without changing timing results.

9 Profiling.

Several routines use *Synchronize*, in particular those of time information. Suppose that a segment of the parallel algorithm has a portion of code in different processors. To obtain partial execution times, this portion can be selected. The routine *BeginProfile (Profile Variable)* called from processor 0 marks the beginning of the portion to be measured; the end is marked by *EndProfile*, also in the proc. 0. In the other procs., the equivalent portions are marked by two calls to *Synchronize*, which must be done in all the processors. If no code corresponding to the parallel algorithm is present, it will simply be called twice. The pair *BeginProfile* - *EndProfile* accumulates execution data in the variable, and then, in any point of the source code, the routine *ShowProfile* will write a report on the output. The returned data are the mentioned CPT, IWT, OWT, NIO, NOO, NTB and TT, detailed by channel if proceeds and desired.

Similar action performs the pair *SynchronizeCPU* - *ShowCPU*, which, using the previous routines, marks the begin and the end of a report, showing data. The main differences are: no accumulation is performed, an implicit profile variable is used, and, when synchronizing, CPU times of all processors are set to be equal to the greatest CPU time. The increased CPU time is accounted as IWT in a virtual channel. Observe that it is a valid approach, due that to obtain the CPU time of a portion of the algorithm, in the real machine, the mentioned message passing strategy needs to be used, obtaining the same results.

As auxiliary routines, *LockCPU* and *UnLockCPU* mark a portion of code in any processor that do not consumes CPU time. None of the simulator routines can be called between them. This is a good mechanism to write on the output without CPU consumption.

10 Implementation details.

The time estimate features are based on the possibility of high precision time measuring in the host computer. The *clock* C function returns the desired times, but usually its resolution is very poor. For example, in a PC, time counter is incremented 18.2 times per second; in most of the Unix systems, time is returned in microseconds, but with intervals of 0.01 seconds. On these situations, the simulator runs, but the time estimate is bad.

So, to set up the simulator in a machine, two routines must be implemented. The routine *SetInstant* must obtain the actual time of day, or CPU time (of the host), and store it in a variable of type *TyInstant*. This type must be defined for convenient storage. Another one, *Interval*, computes the interval between two previous measured instants, returning the difference in microseconds, stored in a long int. The general specification and the implementation for the PC (with $3\mu s$ of precision) of this routines is given in the distribution kit.

11 Experimental results.

In order to experiment with the simulator a set of programs has been prepared. Communication and wait times have been tested with them. Also, several programs test the built topologies. The distribution kit holds a user manual containing a complete example to show the time measuring in several cases. Also, two real programs have been used to experiment: one to calculate the Hessenberg reduction of a matrix, and another to compute the product singular value decomposition (PSVD) of two matrices.

The simulator performance depends strongly on the power of the host processor. Experimental results show that the i80486/33 processor is about twice faster than the T805 transputer running at 20 MHz. The processor of our HP Apollo is 8 times faster than the same transputer.

But the true determinant of the real time needed for the simulation is the complexity of communications. A communication that in the transputer can take few microseconds, in the simulation may involve hundreds of transfers in the process scheduling from the communication routines. The simulated time will be rather exact, but the time employed in the simulation may be many times larger.

To simulate transputer times with precision, a scaling factor is computed by the simulator at run time, to establish the relation of performance between the mentioned transputer and the host processor. Of course, this factor is not very exact, and user can determine it manually, experimenting with true programs. With the correct factor, we have experimented over the mentioned processors to compute the PSVD of matrices with different sizes over different number of processors. Results differ about 10% with the real times. Also, the computed speed up holds this percentage referred to the true one.

In the PC under DOS, Borland C exhibits a very powerful and quick debugger, but stack size is limited to 64kbytes, which reduces the number of processors that can be simulated to nearly 300, with a very small program. In front, under HP UX, lower memory constraints allow big simulations. Here, 4095 processors are started to compute $\Sigma_{i=x}^{y} i$ with a fan-in algorithm implemented over a binary tree of processors, involving about two seconds of simulation.

References

[1] INMOS Limited, “ANSI C toolset user manual”, 1990.

[2] INMOS Limited, “ANSI C toolset language reference”, 1990.