PARSE: Simulation of Message Passing Communication Networks

Eddy Olk
Department of Electrical Engineering
Delft University of Technology
P.O.Box 5031, 2600 GA Delft, The Netherlands

Abstract

The number of design decisions for communication network hardware in message passing distributed memory systems is quite large, as illustrated by many different implemented and proposed designs. Many of the decisions are driven by, on one side, performance requirements of targeted applications for the parallel system, and on the other side, hardware implementation costs. To obtain cost effective communication hardware matching a certain application domain, use of a parallel architecture simulation framework is inevitable.

The simulator presented here, named PARSE, offers a base for such a framework. It can accurately simulate a wide range of communication architectures and, unlike analytic communication models, does properly model all performance aspects. Changing parameters of simulated architectures is very flexible and the object oriented implementation of the simulator facilitates future extensions and enhancements.

1 Introduction

Given the many different communication hardware designs for message passing distributed memory systems, in commercial systems [1, 2, 3, 4], as well as in the research field [5, 6], there seems to be no consensus on the 'ideal' and best communication hardware. Apparently, the design of the communication network in parallel systems strongly depends on targeted applications.

In order to obtain 'optimal' communication hardware design parameters suiting a specific parallel application or application domain, we need to analyze the communication behavior of the application. Simulation of the application is practically the most feasible and flexible solution. This requires a simulation framework as pictured in Figure 1.

The framework consists of a parallel architecture simulator which can run a parallel application, specified in a High-Level Language (HLL), together with its data, on a simulated MIMD architecture. The user communicates

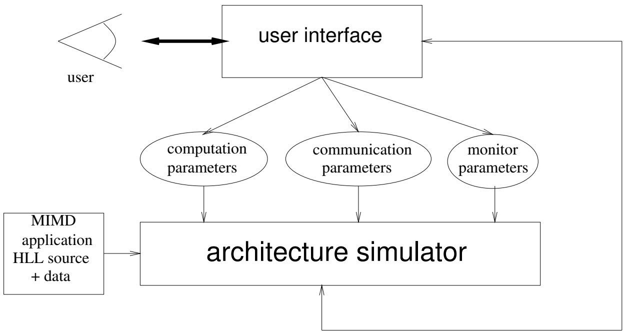

Figure 1: Overview of the simulation framework.

with the simulator using a user interface that shows simulator and application output, allows changing the architecture parameters and lets the user indicate the aspects of the parallel system architecture to be monitored, i.e. the monitor parameters.

Two categories of architecture parameters can be changed: computation parameters, defining the performance of each node in the system (e.g. CPI, Cycles Per Instruction), and communication parameters, defining the architecture and performance of the communication network (e.g. network topology, sizes of buffers). The user can iterate towards an 'optimal' architecture by running the application multiple times with varying parameter values and examine the simulation results.

The framework described above is an extension to our MOVE framework [7] for generation of application specific processors having a RISC, VLIW or transport triggered architecture and broadens the research area to include the matching of MIMD architectures to specific application domains. As a starting point for the development of such an extended framework, a simulator, named PARSE (Parallel ARchitecture Simulator Environment), has been written which simulates a parallel system consisting of a number of processing nodes connected via an interconnection network. The PARSE simulator can run parallel applications written using the SPMD (Single Program Multiple Data) programming model and allows some variation of communication architecture parameters. The remainder of the paper will discuss the design and implementation of this parallel architecture simulator.

Section 2 gives a brief overview of the simulator environment. The communication hardware architecture modeled by the simulator is discussed in Section 3. Section 4 describes the object oriented implementation of the simulator. The performance modeling and the run-time configurable simulator parameters are discussed in Section 5 and Section 6 respectively. Section 7 discusses some results obtained with the simulator and Section 8 concludes the paper.

2 Simulator overview

This section gives an overview of the PARSE simulator. It gives an idea of how the simulator is composed and what it exactly implements. Discussed are the design goals, the simulated parallel architecture, how the simulator is used, and the parallel programming model to be used for writing parallel applications.

2.1 Design goals

The most important design goals considered for the parallel architecture simulator are:

- The simulator should simulate the communication network accurate enough to properly model the simulated hardware. Analytic communication models [8] generally do not model certain performance aspects, like hot spots where a node in the network is congested with communication data.
- In order to iterate towards cost effective communication hardware, the simulator should allow variation of architecture parameters, e.g. network topology, buffer sizes.
- The modeled communication hardware should be suitable for applications with systolic communication patterns, i.e. it should be able to provide (mostly compile time determined) communication channels, possibly with a certain guaranteed bandwidth, between arbitrary nodes.
- The simulator is meant to examine the communicational behavior of applications and is a tool to help obtaining communication hardware suiting a specific application. It is not aimed at providing a tool for performing performance analysis on a specific application in order to tune the application to the underlying hardware, although the simulator may be helpful in this respect.

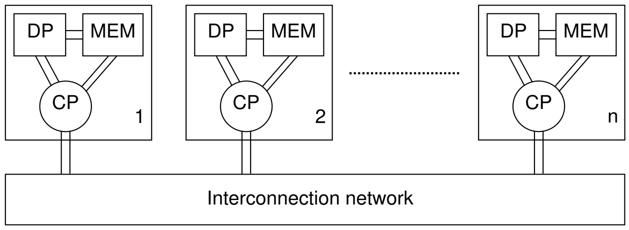

Figure 2: Each node in the network consists of a data processor DP, local memory MEM, and a communication processor CP.

- The simulator should use the *execution-driven* approach which means that it interleaves the execution of the application with the simulation of the multiprocessor architecture. So the simulator simultaneously produces program results, as well as execution statistics from the simulated architecture. This is required since the communicational behavior of an application may be strongly dependent on the specific application's data (which may be provided at run-time by the user) and synthesis of an application may not account for those effects.

2.2 Architecture model

The implemented simulator simulates a parallel system which consists of a number of processing nodes interconnected using an interconnection network. The processing nodes communicate with each other by sending messages over the network. As pictured in Figure 2, each node of the network consists of a data processor (DP), running the application code which (together with the application data) resides in the local memory (M), and a communication processor (CP) that handles the communication, i.e. injection of messages into the network, delivery of messages to the DP, and forwarding of messages not destined for the node in question[1].

2.3 Using the simulator

In order to simulate an application, a simulator executable needs to be created which is constructed of four main components (see Figure 3):

1. **Simulator engine:** the 'core' of the simulator consisting of an event scheduler and event handler. It basically schedules the generated events and handles them in the proper order.

[1] Since the network may consist of a large number of nodes, full connectivity between these nodes is (practically) not feasible and thus messages between non-neighboring nodes must travel along intermediate nodes.

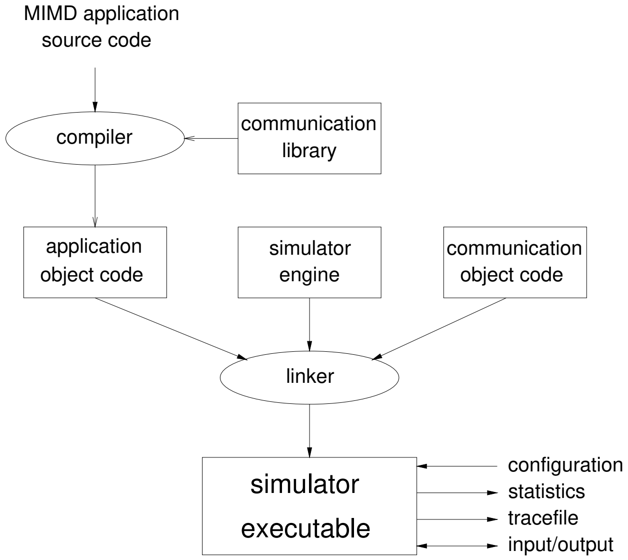

Figure 3: Creation of the simulator executable.

2. **Communication code:** the communication processor code modeling the simulated communication hardware.

3. **Communication library:** communication functions to let the application nodes communicate with each other. The library provides an interface between the application and the communication hardware.

4. **Application code:** code for each processor node implementing the application using the SPMD programming model.

The user only has to supply the last component, the first three components are available in a special archive library file which can be linked together with the application code to get the simulator executable.

Interaction with the simulated application takes place in the usual manner, i.e. via the terminal (standard input and output), application specific files, or the application's user interface. The simulator uses a configuration file to select parameter values and monitor points. The occurring (selected) events are written to a so called *trace file*. After finishing the simulation the simulator prints some execution statistics like (average) link utilization and a list of most communicating nodes. Additional and more detailed information can be obtained by examination of the generated trace file.

2.4 Programming model

Applications that are to be run on the simulator must be written in the C/C++ language using the *SPMD* (Single Program Multiple Data) model. This model implies that every node of the parallel system is running the same program. However, unlike the SIMD (Single Instruction Multiple Data) model, all nodes do not have to execute each instruction of the program simultaneously, i.e. in 'lock-step'. So at a given time, each node may execute a different part of the program.

In order to allow each node to execute a completely different program (despite the SPMD programming model), each node is given a different identification number which can be used by the program to execute a unique program fragment for a specific node. Furthermore, the program can determine the total number of nodes in the system, which is useful for writing scalable (independent of the number of nodes) programs.

Communication between nodes running the SPMD application code is achieved by *communication streams*. These streams closely resemble the I/O streams present in C++. When the application wants to communicate, it creates either an `outstream` or `instream` object which, after successful construction, has established a communication channel to the receiver or from the sender[2] and can then be used for sending or receiving data respectively.

3 Communication architecture

The communication architecture, i.e. the CP model, of the simulator covers a subspace of the whole design space of CPs suiting the design goals listed in Section 2. [9] defines a layered CP design and classification model similar to the OSI model for computer networks. Table 1 summarizes the most important design parameters of the simulator's communication architecture according to this layered CP design model. These parameters are not discussed in detail here, only a brief description of the communication architecture is given.

The data path of the CP implemented by the communication simulator is pictured in Figure 4. The CP has a single *register file* to store data coming over the *input links* from the DP or, via the *physical* links, from neighboring CPs. This register file is multiported so all input links can write to it simultaneously. Also the *output links* can read from the register file concurrently. Within the register file, each input link has (and manages) its own register space to store its incoming data. The FIFO (First In First Out) buffers positioned between the DP and the CP improve performance by allowing the DP to read/write small amounts

[2] Actually, there may not be a complete communication path between sender and receiver, e.g. when sending data the sender may actually have closed the communication stream before the data has actually arrived at the destination.

Table 1: Important layer design parameters of CP architecture implemented by the simulator.

parameter	**values**
transport layer	
entity representation	
addressing mode	absolute
address field	variable size
data field	variable size
network layer	
entity representation	
addressing mode	absolute
address field	variable size
data field	variable size
buffering	centralized
routing	
path control	network
path	deterministic
collision resolvement	buffered blocking
network support	
degree	$1 - n$
topology	multidimensional meshes
	k-ary n-cubes
deadlock resolvement	acyclic
livelock resolvement	fair sharing
data link layer	
virtual layer support	yes
entity representation	
data field	8, 16, 32, 64 bits
flow control	
granularity	tight
window	≥ 1

of data fast without the possibility of blocking for every data element read/written.

In order to be able to make more effective use of the available bandwidth of each link, and help avoiding deadlock problems, every (physical) link is divided in *channels* or *virtual links*. So messages can be sent (independently) over each channel and all channels must share the bandwidth of the single physical link.

To process the incoming communication data, an input link has a *router* for each channel of the input link. The router examines (the address part of) an incoming message and decides where this message must be routed to, i.e. to one of the output links or to the DP. The router then tries to allocate a free, unused, channel from the specific output link. When this succeeds, the router will keep the output link informed of the presence of message data for the channel. The output link will read the message data from the register file and send it over the (physical) link, thus forwarding the message.

The PARSE simulator uses the following message format (see Figure 5) where a message consists of three parts:

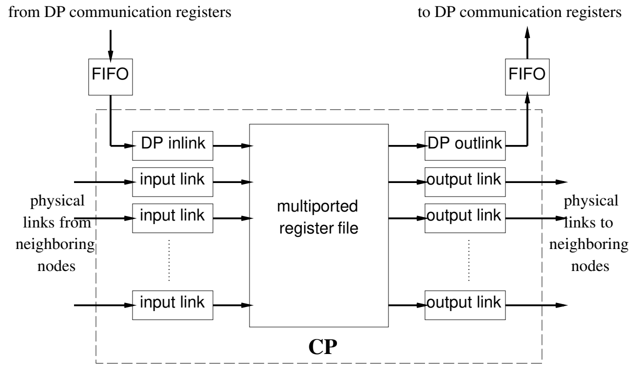

Figure 4: Datapath of CP model used by simulator.

1. **Address:** the destination address of the message.
2. **Data:** the message's contents, i.e. the actual application related data.
3. **EOM:** End Of Message, indicates the end of the message.

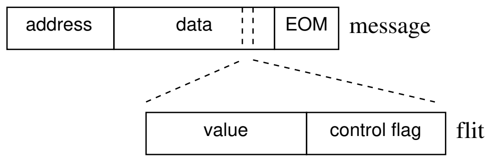

Figure 5: Message and flit formats.

Each of these parts consists of one or more *flits*. A flit is simply a data unit which can hold a certain amount of data depending on the targeted communication architecture and it is the smallest unit on which flow control is performed. The simulator supports flits that can hold 8, 16, 32, or 64 bit values. Besides the *value* of a flit, each flit also has a *control flag* of one bit which allows distinguishing two types of flits: *control flits* (such as EOM) which have the flag bit set, and *data flits* which have the flag bit reset. Note that the size of a flit defines the data width of the internal data paths of the CP, e.g. the size of each register in the register file.

4 Object oriented implementation

The simulator is implemented in an object oriented manner and written in the language C++. It is event driven and its strategy is plain simple, as shown in Figure 6: a number of threads, modeling the simulated architecture, execute

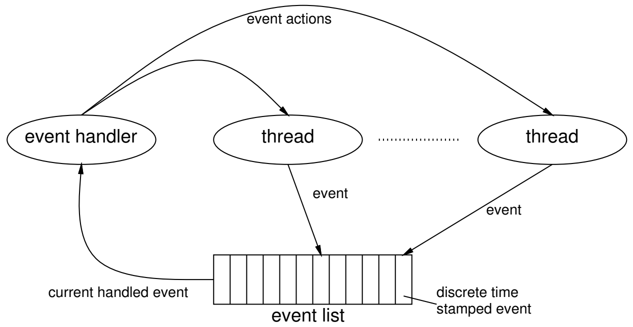

Figure 6: Strategy of the simulator.

in parallel and will generate discrete time events that are placed in a (time-sorted) event list and eventually handled by an event handler.

Each thread can be in one of the following states: *created*, the thread has just been created, *runnable*, the thread is running or ready to run, *waiting*, the thread is waiting for a so called *resource* to become available, or *terminated*, the thread is finished. When no more events are generated, i.e. no threads are runnable anymore and thus all threads are terminated and/or waiting for a resource, the event handler becomes active and takes the event with the earliest time-stamp from the event list. The event handler then executes the corresponding event action which may result in more events being generated and threads becoming runnable again (due to resources being freed). When no more events are present in the event list and no threads are runnable, the simulation is finished and the simulator exits.

The object oriented implementation of the simulator is pictured in Figure 7. Note that although the current implementation of the simulator executes on a single processor, one of the motivations for the object oriented organization is that it forms a good base for future parallel implementations executing on multiple processors. At the top level, the simulator consists of four objects:

1. **Configuration:** responsible for the run-time configuration of the simulator (see Section 6).

2. **EventList:** the time sorted list of future events which need to be handled.

3. **EventHandler:** handler which gets the current event from the event list and executes the corresponding action.

4. **Network:** the network of processing nodes modeling the simulated parallel architecture.

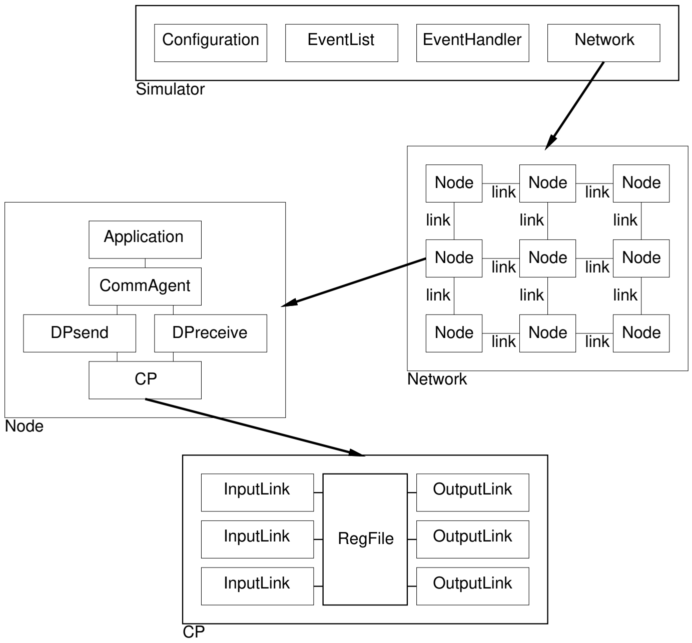

Figure 7: Object oriented implementation of the simulator.

Modeling of the parallel architecture is very suitable for an object oriented approach and the network object is composed of objects similar to the way the simulated parallel architecture is composed of various hardware modules. The `Network` object will create the desired network topology, as indicated by the `Configuration` object, by constructing it from `Node` and `Link` objects which respectively model a network node and a physical link between two nodes in the network.

Likewise, the `Node` object consists of the following objects:

- **Application:** runs the SPMD application code.

- **CommAgent:** interface between the application code and the communication hardware. It provides the application code with communication streams.

- **DPsend:** models hardware for injection of messages from the DP into the network (CP).

- **DPreceive:** models hardware for delivery of messages from the network (CP) to the DP.

- **CP:** models the architecture of the communication processor.

To model the communication architecture, as shown in Figure 4, the `CP` object is composed of a `RegFile` object, which models the register file, and several `InputLink` and `OutputLink` objects, modeling the input and output links respectively.

Table 2: Performance modeling parameters.

processing performance	communication performance		
	bandwidth	latency	
		routing	flow control
processor cycle time	consecutive flit transfer time	buffer read time buffer write time link delay time	control crossbar delay time flow link delay time flow update time

5 Performance modeling

This section discusses the performance modeling of the simulated parallel architecture and gives the parameters required to be able to specify the performance. These parameters are summarized in Table 2. The performance modeling falls apart in two categories: performance of the data processor and performance of the communication processor, i.e. the communication network.

5.1 Processing performance

Although PARSE is aimed at analyzing communication performance and the specific architecture of the DP is not of special interest to the simulator, the performance of the DP is important since it influences the communicational behavior of the application. Yet, we do not want to model the DP too accurately since this unnecessarily slows down the simulation. So we want to model the processing performance just accurate enough to yield the proper communicational behavior.

PARSE obtains fast application simulation speeds by simply executing application code directly on the processor running the simulator executable. The execution time of the computations is modeled using a technique called *code augmentation* (as described by [10]). With code augmentation instructions are added to the application code in order to time the code.

Contrary to the PROTEUS simulator [11] that uses a separate program to augment the assembler output file, PARSE has code augmentation implemented at the compiler level. The GNU C/C++ compiler has been modified to produce augmented code when its basic block profiling switch is enabled. The current implemented augmentation assumes that each instruction takes one clock cycle to execute and that memory accesses do not take additional cycles, there are no cache misses for memory references. Given that the framework is meant to analyze specific applications, or application domains, these assumptions are acceptable.

At the start of each *basic block*[3] of the application code, a counter is incremented with a value representing the execution time of the specific basic block in clock cycles. When the application needs to communicate (send or receive) with another node, this counter is considered and the node in question is delayed for the time represented by the counter before the actual communication action takes place (and the counter is reset to zero).

So application code of all nodes may execute out of phase with respect to the moment in time but will synchronize as soon as they need to communicate. This strategy is sufficient for applications using synchronous message passing where communication takes place at ‘known’ points in the application code, and no limit needs to be put on the amount of time the nodes may execute out of phase. However, with applications using asynchronous message passing, messages may arrive, be delivered, and processed even when the receiving node was not waiting for an incoming message. Then such a limit is required to avoid unrealistic simulation results since an incoming message may initiate other messages to be sent or the flow of the application program to be changed.

The simulator supports applications using asynchronous message passing by checking a (user defined) threshold value at the start of each basic block. A low threshold value will keep the executing nodes close in time but will reduce simulation speed due to the overhead involved in the additional context switches of the threads modeling the application. A high value may result in faster simulation by allowing nodes to execute more out of phase.

As an example of the implemented code augmentation, Figure 8 shows the pseudo code for the additions made to each basic block together with the actual added assembler code output for the HP-PA RISC family processors. Note that to minimize the overhead of the augmentation code, a register (`r16`) has been allocated[4] to hold the augmentation counter and that the counter counts down (it is initialized with the threshold value). The first added in-

[3] A basic block is sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end [12].

[4] The compiled code will not employ the r16 register at all, the r16 register is used exclusively throughout all the code to hold the value of the cycle counter.

pseudo code:

```
cycle count = cycle count + basic block cycles
if (cycle count > threshold) then
   delay for cycle count X cycle time
   cycle count = 0
endif
```

added code

```
actual basic block statements
```

HP−PA RISC assembly code:

```
    ldo    -4(%r16),%r16    ;r16 = r16 - 4
    comibt,<,n 0,%r16,L9999 ;if r16>0 branch to L9999
    stwm %r2,384(0,%r30)    ;create stack frame,save r2
    stw    %r1,-68(0,%r30)  ;save r1
    stw    %r26,-72(0,%r30) ;save r26
    stw    %r28,-76(0,%r30) ;save r28
    stw    %r29,-80(0,%r30) ;save r29
    bl    _augment_delay,2  ;perform delay
    nop                     ;delay slot
    ldw   -68(0,%r30),%r1   ;restore r1
    ldw   -72(0,%r30),%r26  ;restore r26
    ldw   -76(0,%r30),%r28  ;restore r28
    ldw   -80(0,%r30),%r29  ;restore r29
    ldwm -384(0,%r30),%r2   ;restore r2
L9999
    ; actual basic block code
```

Figure 8: Augmentation code, pseudo code and HP-PA RISC assembler output.

struction subtracts the execution cycles of the basic block from `r16`. The second instruction will jump to the original basic block if the result of the subtraction is still positive. If on the other hand the threshold is reached, the registers that are 'live'[5] are saved on the stack (this minimizes the context switch overhead) and a function call is made to delay the node. The `augment_delay` function delays the node for the number of cycles present in the cycle counter `r16` and resets it to zero. After the delay, the previously saved registers are restored and the instructions of the basic block are executed.

As is clear from Figure 8, a lot of code is added for each basic block. However, most of the time only the first two instructions are executed. If we assume an average basic block length of 6 instructions and an average cycles per instruction (CPI) of 1, the slowdown due to the code augmentation is roughly 25%. Simulating the architecture of the DP is most likely to give worse results.

Besides modeling the performance of the application using code augmentation performed by the compiler, PARSE allows that the user inserts code timing statements into his code. Either way, by using a parameter to specify the cycle time of the processor, the actual (simulated) execution time of the application code can be determined.

5.2 Communication performance

The communication performance falls apart in two attributes: bandwidth and latency. Bandwidth is the amount of data that can be sent along a link or node in a certain time. Latency is the time data takes to move through a node or traverse a link.

The link bandwidth is determined by the size of a data transfer and the minimum required time between two data transfers over a link. Since a flit is the minimal data unit on which flow control is performed, the peak link bandwidth is fully defined by specifying the size of a flit and the minimal time between two consecutive flit transfers over a link. The CP architecture used by the simulator is pipelined in such a way that the (internal) bandwidth of the CP architecture matches (equals) that of its links. The bandwidth is thus fully defined by a flit size parameter and a parameter indicating the minimum time between consecutive flit transfers over the physical link.

The latency is determined by far more timing parameters (see also [13]). We can divide the latency aspects in: routing latency, the time taken by data to move from one node to another, and flow control latency, the time taken by flow control information to go from one node to another (in the opposite direction of the corresponding data).

For the architecture implemented by the simulator, the routing latency t_{rl} of a flit (starting when a flit comes in over a link) is given by

$$t_{rl} = t_{bw} + t_{rd} + t_{ca} + t_{la} + t_{br} + t_{ld}$$

So the routing latency is composed of:

- **Buffer write time** (t_{bw}): the time required to write the incoming flit in the buffer associated with the link (and allocated in the register file).
- **Router decision time** (t_{rd}): the time required by the router to decide where the incoming data must be routed to. This time strongly depends on the employed routing algorithm and message address format. Obviously, after the routing direction is determined, all following flits in the message do not require any routing decision time.
- **Channel allocation time** (t_{ca}): the time taken to allocate an unused channel of the output link the flit must be routed to. This time is only applicable for the first flit of a message and it depends on whether there are any free channels available.
- **Link arbitration time** (t_{la}): the time taken before the flit is granted access to the link. This is dependent on other flits concurrently competing for link bandwidth.
- **Buffer read time** (t_{br}): the time needed to read the flit from the buffer (register file).
- **Link delay time** (t_{ld}): the time taken by the flit to traverse over the link and arrive at the next node. This

[5]In use at that moment to hold application data, return link pointers, function arguments, etc. In the fragment of Figure 8 the registers r1, r2, r26, r28, and r29 are live.

time includes any time possibly required for the flit to synchronize with the next node.

Of these times composing the routing latency t_{bw}, t_{br}, and t_{ld} are constant and may be used as performance parameters. On the other hand, t_{rd}, t_{ca}, and t_{la} are variable (their value is dependent on run-time conditions) and not suited as performance parameters.

The flow control latency t_{fcl} (starting when a flit has just been read from the register file) is given by

$$t_{fcl} = t_{ccd} + t_{fla} + t_{fld} + t_{fu}$$

It thus consists of:

- **Control crossbar delay time** (t_{ccd}): the time taken by the flow control passing the control crossbar switch (allowing flow control info to be exchanged between each output link and with every input link) to the input link managing the register file subspace the corresponding flit was read from.

- **Flow link arbitration time** (t_{fla}): time taken before the flow control can be sent over the flow control link to the previous node due to competing flow control transfers for the same link.

- **Flow link delay time** (t_{fld}): the time taken by the flow control information to arrive at the previous node, including possible synchronizing delay. Note that the effect of this parameter is usually rather small due to the use of flow control window (see [9]).

- **Flow update time** (t_{fu}): the time needed to update the flow window of the previous node, in order to properly represent the buffer space status on the receiving side.

For the flow control latency there is only one time that is variable: t_{fla}. The other times t_{ccd}, t_{fld}, and t_{fu} have a constant value and can be used as performance parameters.

6 Run-time configuration

To avoid needing to create a new simulator executable every time some parameter is altered, the simulator allows run-time configuration of quite a number of parameters using a configuration file. The configurable parameters fall apart in three categories: (communication) architecture configuration, architecture performance parameters, and configuration of the monitor points to examine communication activities.

6.1 Architecture configuration

The following parameters of the communication architecture can be configured using the configuration file:

- **Size and topology of network.** The simulator currently covers most topologies by supporting multi-dimensional mesh networks and k-ary n-cubes.

- **Routing strategy.** Dependent on the selected network topology, a routing algorithm can be chosen appropriate for the specific topology. Default a dimension routing algorithm is used (similar to E-cube routing).

- **Number of virtual channels.** The number of virtual channels of each physical link, as well as of the links from/to the DP to/from the CP, can be set to a value ≥ 1. The number of virtual channels between DP and CP defines the maximum number of communication streams that may be used (opened) simultaneously.

- **Buffer sizes.** The amount of buffer space available, i.e. reserved in the register file, for each channel of an input link can be specified. Buffer sizes of each channel of an input link need not be the same. Also the sizes of the FIFOs between the DP and CP (see Section 3) are configurable.

- **Flit size.** The size of a flit can be specified to have, in addition to the single control bit, 8, 16, 32, or 64 bits to hold the flit's value. As already mentioned in Section 3, the flit size is tied to the data path width of the architecture.

6.2 Architecture performance

To allow varying the processing and communication performance, the simulator supports configuration of the following performance parameters (all time parameters are specified in the minimal time unit[6]):

- **Processor cycle time:** the clock cycle time of the processing nodes.

- **Link delay time:** the time needed for a flit to propagate over a physical link.

- **Flow link delay time:** the time taken by flow control information to travel over a physical link.

- **Link bandwidth:** the bandwidth of physical links as well as the bandwidth between the DP and the CP can be specified as the number of time units that must be present between two consecutive flit transfers.

[6] It is up to the user to define what a time unit represents exactly and thus what the actual time resolution is of the simulation (e.g. 1 μs).

- **Buffer access time:** the time required to access the register file, i.e. to read/write a flit from/to a register in the register file.

Although generally not necessary for application analysis, the user can also specify the additional parameters given in Table 2 which are not mentioned above. These additional parameters are however rather architecture specific and their influence on the overall performance is relatively small compared to alteration of the parameters listed above.

6.3 Monitoring & tracing

Besides defining the architecture and its performance, the configuration file can also contain commands to specify the monitoring of various events performed within the communication network. Default the simulator will not monitor any events but when monitoring of one or more events is enabled, the simulator will generate a trace file containing a time sorted list of all requested events, i.e. the *traces*. Detailed communication statistics can then be determined by examination of these traces using data analysis tools.

Monitoring of specific events can either be activated *globally*, for each node in the network, or *locally*, for a specific node or range of nodes. Table 3 summarizes the implemented monitor parameters.

7 Discussion

To evaluate the current simulator implementation, work is in progress on a number of case studies of different parallel applications. These applications include simple ones, like matrix multiply and Fast Fourier Transform (FFT), but also more sophisticated ones, like the traveling salesman problem and parallel VHDL simulation.

Based on the experiences with these simple case studies, the simulator will be refined and extended. For instance, in the future the architecture will be complemented with its own access path to the local memory (as already shown in Figure 2). This allows the CP to deliver and send communication data independent of the DP, thus reducing the communication overhead by overlapping computation and communication.

As an illustration of the use of PARSE, Figure 9 shows some results obtained for a straightforward implementation of a parallel FFT algorithm [14]. A number of simulations were performed for FFT of a vector of 4096 elements on a mesh topology network with 4 virtual channels per link and a dimension routing algorithm. During the simulations, the network size and the bandwidth of the physical links were varied.

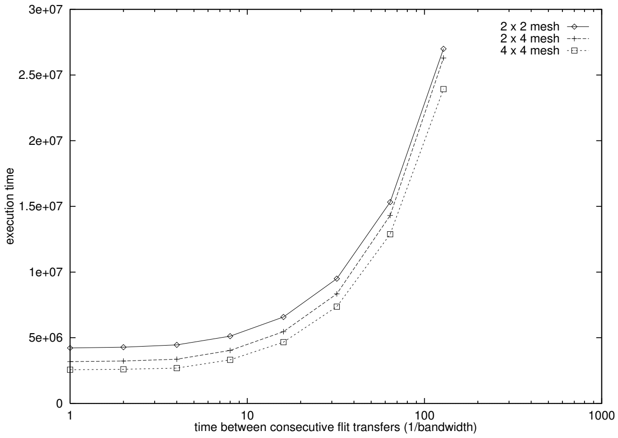

Figure 9: Execution times of parallel FFT of a 4096 element vector for different network sizes and link bandwidths.

In Figure 9 we see that, starting with the maximum link bandwidth, decreasing the link bandwidth does not significantly increase the total execution time of the program until it is reduced with a factor of about 10. Then the communication time starts to become a major part of the total execution time. This allows the user to make a trade-off between execution time and link bandwidth which is important since the bandwidth of a link is generally linear with the number of package pins and reducing the number of required pins allows the use of cheaper chip packages.

Another point that can be digested from Figure 9 is that the specific parallel implementation does not scale very well since increasing the number of processing nodes from 4 to 16 only reduces the execution time with about 30%. This is largely due to the fact that the implementation uses synchronous message passing and more communication needs to take place since there are more processing nodes but the problem size has stayed the same. So in this case it might be advantageous to reconsider the implementation and make use of asynchronous message passing so communication time and computation time may be overlapped.

8 Conclusions

The PARSE simulator for simulation of message passing communication networks was presented. The simulator is aimed at communication analysis of parallel application domains in order to find the most cost effective communication hardware configuration matching the requirements of the specific domain. Its strong points here are the flexible configuration of architecture, performance, and monitor parameters together with the support for systolic communication. In addition, the object oriented organization of PARSE makes it easy to extend its features as well as work

Table 3: Possible monitor points.

ostreamopen	An output stream is opened.
ostreamclose	An output stream is closed.
istreamopen	An input stream is opened.
istreamclose	An input stream is closed.
linkflit	A flit starts to travel over a physical link between two nodes.
linkack	An acknowledge (of flit receipt) starts to travel over a physical link.
regrd	A flit is read from the register file.
regwr	A flit is written to the register file.
dpcpwr	A flit is written to the (inject) FIFO between DP and CP.
dpcprd	A flit is read from the (inject) FIFO between DP and CP.
cpdpwr	A flit is written to the (eject) FIFO between CP and DP.
cpdprd	A flit is read from the (eject) FIFO between CP and DP.

towards a parallel implementation in order to reduce simulation times. Its accurate simulation of the architecture avoids missing possible performance bottlenecks like hot spots which are generally not considered by analytic communication network models.

Acknowledgements

Grateful acknowledges to Henk Corporaal for support and comments, Arlet Ottens for providing the thread mechanism and Wilco van Hoogstraeten for helping stress test the simulator.

References

[1] S. Borkar et al. iWarp: An Integrated Solution to High-Speed Parallel Computing. In *Proceedings of Supercomputing '88*, pages 330–339. ACM SIGARCH, January 1989.

[2] Daniel Hillis. *The Connection Machine*. MIT Press, 1985.

[3] Thinking Machines Corporation. Connection Machine CM-5 Technical Summary, November 1992.

[4] INMOS Limited. The T9000 Transputer Products Overview Manual. INMOS Databook series, April 1991.

[5] William J. Dally. *A VLSI Architecture for concurrent Data Structures*. Kluwer Academic Publishers, 1987.

[6] H. Corporaal and J.G.E. Olk. A Scalable VLSI MIMD Routing Cell. In *6th Distributed Memory Computing Conference Proceedings, Portland*, pages 654–661. IEEE Computer Society Press, April 1991.

[7] Henk Corporaal and Hans (J.M.) Mulder. MOVE: A framework for high-performance processor design. In *Supercomputing-91*, pages 692–701, Albuquerque, November 1991.

[8] Laxmi N. Bhuyan, Qing Yang, and Dharma P. Agrawal. Performance of Multiprocessor Interconnection Networks. *IEEE Computer*, pages 25–37, February 1989.

[9] Eddy Olk and Henk Corporaal. The OSI Model applied to MIMD Communication Processor Design. In *ParCo 93: Conference on Parallel Computing, Proceedings of the International Conference*, September 1993.

[10] P.J. Weinberger. Cheap dynamic instruction counting. *AT&T Bell Laboratories Technical Journal*, 63(8), October 1984.

[11] Eric A. Brewer. Aspects of a Parallel-Architecture Simulator. Technical Report MIT/CS/TR-527, MIT, February 1992.

[12] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, principles, techniques, and tools*. Addision-Wesley Publishing Company, 1986.

[13] Andrew A. Chien. A Cost and Speed Model for k-ary n-cube Wormhole Routers. In *Hot Interconnects 93*, August 1993.

[14] Geoffrey C. Fox et al. *Solving Problems On Concurrent Processors: General Techniques and Regular Problems*, volume 1. Prentice Hall, 1988.