Simulating Parallel Architectures with BSPlab

Lasse Natvig

Dept. of Computer and Information Science, Norwegian University of Science and Technology, Gløshaugen, N-7491 Trondheim, NORWAY
Lasse.Natvig@idi.ntnu.no

Abstract. BSPlab is a simulation environment for studying the interplay between hardware and software in parallel computing. It offers the BSPlib parallel programming library and is based on Bulk Synchronous Parallel (BSP) computing [1], [2]. BSPlab contains a set of high-level performance models of parallel architectures. It can be used as a tool for architectural level design space exploration of BSP computers both in research and teaching. The paper introduces BSP, BSPlib and BSPlab. Then it presents the architectural models and their parameters, and discusses how they can be used to make experiments that show how the different aspects of a computer affect the performance of an application.

1 Introduction

Although it has been claimed an ongoing convergence in parallel computer architecture [3] — understanding the interplay between hardware and software in parallel computing is still difficult. This understanding is crucial both for developing efficient parallel applications on contemporary supercomputers and for designing competitive high-performance parallel computers. An important way of studying the HW/SW interplay in such complex parallel computing systems is by simulation. The BSPlab environment was developed for use in research and teaching of parallel computer architectures as well as for performance studies of parallel applications.

BSPlab offers parallel application development using *BSPlib* that is a standardised library for parallel programming based on the BSP (Bulk Synchronous Parallel) model [4]. Libraries such as PVM and MPI have a larger user community, but we believe the simplicity of BSPlib and the generality of BSP makes it more suitable for developing applications that are portable to a large variety of parallel computer architectures without rewriting code to maintain efficiency. BSPlab contains architectural performance models for evaluating the efficiency of such portable software.

The paper starts by giving a short introduction to the BSP model and the BSPlab environment in Section 2. Section 3 gives a thorough presentation of the parameters in the set of architectural models available in BSPlab. This gives the background for understanding what kind of architectural experiments are possible. In Section 4 we summarise briefly the status of BSPlab with respect to test programs and conducted

experiments. Section 5 ends the paper by presenting experience, current work and a few concluding remarks.

2 A Short Introduction to BSP and BSPlab

Leslie Valiant proposed the Bulk Synchronous Parallel (BSP) model in 1990 [1]. The BSP model is a theoretical framework outlining how parallel computations can be organised in a way that bridges the gap between the needs of the programmers and the computing hardware offered by the computer architects and designers.

The model is defined as the combination of three attributes: *i)* a number of *components* performing processing and/or memory functions, *ii)* a *router* that delivers messages point to point between pairs of components, and *iii)* a *synchronisation facility* that is able to synchronise all or a subset of the components at regular intervals. A computation is described as a sequence of *supersteps*. During a superstep, the components perform computations asynchronously, and the synchronisation facility guarantees that all components have finished the current superstep before they proceed to the next one.

William F. McColl has been central in bringing the BSP theory to practical programming. His paper on Bulk Synchronous Parallel Computing [2] describes how the performance of BSP computers can be characterised by only four parameters. Informally these are processor speed (*s*), number of processors (*p*), synchronisation cost (*l*) and global computation/communication balance (*g*). Hill has reported the values of these parameters for the most popular supercomputers [5]. The BSP parameters are central in algorithm analysis for BSP applications, and make it possible to develop parallel programs with *portable* efficiency.

BSPlab is an environment for experimenting with BSP programs on a rich variety of parallel computer architectures. Parallel applications, benchmarks or specialised performance testing programs are written in C or C++ using BSPlib for communication and synchronisation. The user may select among various predefined parallel computer architecture models or might define her own architecture. The BSP programs are then debugged and executed in the BSPlab environment to achieve the measures specified by the user to support the current foci of the performance study. A typical goal can be a better understanding of the interplay between the selected BSP-architecture and the parallel program. In short, BSPlab can be regarded as an experimental tool for architectural level *design space exploration* of BSP computers.

BSPlab uses the standard Microsoft Developer Studio for the C++ programs. A consequence is that it has a good programming environment that also can be used for developing BSPlib applications that are targeted for real parallel computers.

BSPlab was developed at NTNU in Trondheim, Norway, as the diploma work of Dybdahl and Uthus [6]. It is built upon the process-oriented discrete event simulation package C++SIM [7]. Some ideas and methods were also found in MultiSim++ [8]. This is a simulation package for simulating processing elements connected in different networks. BSPlab is available from `www.idi.ntnu.no/bsplab`.

3 BSPlab Architectures and its Parameters

This section describes the BSPlab models of different parallel computer architectures. When BSPlab starts execution, it loads an *architecture definition* file that selects a model and gives the values of its parameters. Knowing the parameters is necessary for understanding what kind of experiments the models can be used for.

3.1 Overview of Available Architectures

BSPlab currently includes models of two abstract architectures and three real architectures. The abstract architectures are useful for teaching BSP programming, and for studying performance of applications running on parallel machines that give little and therefore easy-to-understand influence on the measured performance. Experiments on such simplified architectures may provide a useful step for understanding the more complex effects that more realistic architectures have on application performance.

The main difference between the three real architectures is their communication medium and interface for exchanging information between the processors. The distributed shared memory architecture uses a shared bus, the tightly coupled multiprocessor architecture uses dedicated communication links and the NOW architecture uses a computer network such as Ethernet.

- **Null machine** This abstract machine models an ideal parallel computer machine where both communication and synchronisation is free. However, the computation time used by the processors can be modelled in several ways. See Section 3.2.
- **Simple machine** The simple machine model extends the null machine by offering simple means of representing the time used for communication and synchronisation. It is further described in Section 3.3.
- **Distributed shared memory (DSM)** This architecture is functionally like a multiprocessor with the processors connected through a single common bus to a shared memory. The *logically* shared memory is *physically* distributed among the processors. Each processor can access data in own local memory, in shared memory stored locally, or in shared memory stored in other processors. The parameters for this architecture models the speed of the common bus and the method of synchronising the processors. The model is further described in Section 3.4.
- **Tightly coupled multiprocessor (TCM)** This is a *family* of architectures where the processors are interconnected through one of a set of different network topologies. Parameters are available for specifying topology, network size, the communication between the processor and the network, and methods and performance of communication by messages. It is elaborated in Section 3.5.
- **Network of workstations (NOW)** This model represents a set of workstations or PCs interconnected in a local area network (LAN). The Ethernet is chosen as network standard and the model contains 10 parameters that specify the BSPlab Ethernet implementation. In addition, it has parameters for specifying the simulation

of the network traffic generated by other applications using the same network. This BSPlab model is given a brief description in Section 3.6.

- **User defined machines** The original BSPlab documentation [6] contains a short description of how the advanced user may implement other architectural models than the preimplemented ones. That possibility will not be elaborated here.

3.2 Null Machine

When executing BSP applications on the null machine both inter-processor communication and synchronisation ”takes no time”. Still, the model has *general* parameters that make it useful for learning fundamental aspects of parallel application performance. These are summarised in Table 1.

A fundamental choice in BSPlab is between manual and automatic timing. With *Manual timing* the programmer manually inserts “hold-calls” into the program to specify the time consumption of various program segments. It corresponds to advancing the simulation clock in discrete event simulations. It is useful in experiments where you want to highlight the performance effects of certain parts of an algorithm, or when using coarse grained time modelling. An example is cases where you want to verify analytically derived performance expressions by simulation experiments [9].

Automatic timing means that the BSPlab environment measures the time used to execute the various parts of the program. The programmer is then relieved from the burden of inserting the hold calls. In addition, the time modelling will be more accurate and realistic. When discussing timing in BSPlab it is important to distinguish the processor speed of the *virtual processors* in the BSP computer being simulated from the real *executing processor* in the PC executing the simulation model. A problem with automatic timing is that running the same BSPlab experiment on a fast executing processor will give better performance results than on a slower one. There are cases where this is unpractical, and the parameters InternalCPUbenchmark and ThisCPUSpeed can be used to avoid it.

InternalCPUBenchmark[1] is a parameter that turns on a small benchmark program that is run initially to estimate the speed of the executing processor.

Table 1. Parameters for the *null machine* BSPlab model

Parameter name	*Informal description*
NumberOfProcessors	No. of processors in BSP computer
AutomaticTiming	Turns on automatic timing of code
ManualTiming	Turns on manual timing of code
InternalCPUBenchmark	Estimate speed of executing processor
ThisCPUSpeed	Set speed of executing processor
VirtualCPUSpeed	Set speed of virtual BSP-processors

[1] The original name for this parameter is *AutomaticCPUTiming* [6]. However we have changed it to better reflect its meaning and to avoid confusion with *AutomaticTiming*.

If a BSPlab user has a set of completed experiments and upgrades to a more powerful PC, she might run an experiment with the InternalCPUBenchmark parameter set and write down the reported speed of the new executing processor. By using this value for the *ThisCPUSpeed* parameter she might carry on with new experiments using automatic timing that are comparable with the older results. More details on this can be found in Line's Diploma Thesis [10].

The parameter *VirtualCPUSpeed* is used to specify the speed of the virtual processors relative to a 100MHz Intel 486. The parameter is useful for scaling the time modelling so that it is representative for the BSP computer we have in mind. As an example, when we are modelling a BSP computer using 500MHz Pentium IIIs as computing nodes the parameter should be set to 4.2.

Automatic timing can also be combined with manual timing. In the first stage, automatic timing may be used in repeated experiments to produce average values that can be used as good estimates for realistic time consumption in the main program segments. In the second stage, we switch to manual timing with these values used as parameters to hold-calls. Then we might in many cases obtain both the realism of automatic timing and repeatedness and faster execution characteristic for manual timing.

The null machine contains few and only very basic parameters. However, studies of non-trivial applications should often start with simple experiments with only a few parameters being varied. Also, the ideal computer offered by the null machine, with free communication and synchronisation, gives an easy way to derive an upper limit for the performance that can be obtained by a given parallel program.

3.3 Simple Machine

As the name indicates, the *simple machine* BSPlab model is also an unrealistic machine model that simplifies a lot. As all BSPlab models, it contains the parameters of the null machine. In addition, it has the two parameters shown in Table 2 for specifying a linear time model for synchronisation or communication. The BSPlab user sets both to appropriate floating-point values. The parameters have the prefix "Simple_" to indicate that it is only available for this model. The three more realistic models have more comprehensive ways of specifying the cost of communication and synchronisation.

If the parameter *Simple_TimeToSynchroniseOneProcessor* is set to a non-zero value the BSP-processor will first perform a free barrier synchronisation, and then all the processors used, assume the number is *N*, waits by doing a "hold" of N multiplied with the value of the parameter. This means that the time used on the synchronisation is constant for a given *N*. A barrier synchronisation will often depend on the distribution of the "arrival times" when the various processors start on the synchronisation process. A consequence is unpredictable synchronisation time. However, the linear cost model used in the simple machine eliminates this unpredictability and might therefore make it easier to understand the performance of complex parallel programs.

Table 2. Parameters for the *simple machine* BSPlab model

Parameter name	*Informal description*
Simple_TimeToSynchroniseOneProcessor	Linear synchronisation cost
Simple_TimeToSendOneByte	Linear communication cost

Similarly, the parameter *Simple_TimeToSendOneByte* is a floating-point value multiplied with the number of bytes sent by a processor to calculate the time used to perform the send operation. Note that the time used is independent of what any of the other processors are sending. Again, this is a simplification since many communication structures will reach a point of saturation where communication is slowed down by other traffic.

If further simplifications are needed to increase understandability, an option is to set one of the parameters to zero. For instance, the experimenter can focus entirely on the communication of an application by letting synchronisation and computation be free. Note also that BSPlab offers more than 20 *logging parameters* that are used for reporting. As examples, the experimenter may log the execution time used by a superstep, the time used on synchronisation, and the number of bytes communicated by BSMP or DRMA.[2] The values may be reported by superstep and by processors, as average values and as grand totals. The logging parameters are available for all BSPlab machine models [6].

A general linear communication cost model can be expressed as $T = a + b \times n$ where T is the communication cost, a is the startup cost, b is the time used to send one byte, and n is the number of bytes sent. The linear model used in the BSPlab simple machine has $a = 0$ and thus fails to model the startup cost that is typical for many communication mechanisms. Section 7.7 in [6] describes how to make a user defined machine model with this extension and how to specify an alternative way of modelling the time used on synchronisation.

3.4 Distributed Shared Memory (DSM)

The BSPlab model for Distributed Shared Memory (DSM) machines has 10 parameters, see Table 3. We start by explaining the first five parameters that are common for the DSM, Tightly Coupled Multiprocessor and the Network of Workstations. The parameter *BytesInGetMessage* specifies the size of the message that implements a call to the BSPlib routine `bsp_get`. A processor uses this routine to request data from a remote processor. Similarly, *BytesInSyncMessage* is the size of one synchronisation message. The processors send a multitude of such messages to perform a barrier synchronisation. BSPlib offers two modes of communication called Direct Remote Memory Access (DRMA) and Bulk Synchronous Message Passing (BSMP) [4]. In BSMP the message must be marked and the extra number of bytes used for this is given by *BSMPMessageMarkSize*.

[2] BSMP and DRMA are two ways of communicating in BSPlib, see Section 3.4.

In the DSM, TCM and NOW models it is assumed that the processor is transferring the data to a *communication controller* in the same node before this controller dispatch it on the communication medium. *TimeToStartPacketizing* and *TimeToPacketizeOne-Byte* implement a linear communication cost model of the kind discussed at the end of the previous section. In addition to this comes the time used for transmitting the data on the shared bus which performance is modelled by three parameters. *Processor-Bus_Width* gives its width in bits and gives together with *ProcessorBus_Frequency* (in Hz) the bandwidth of the bus. The parameter *ProcessorBus_ArbitrationGap* specifies the time used to specify who is the next processor to have control of the shared bus.

Table 3. Distributed Shared Memory (DSM) architecture parameters

Parameter name	*Informal description*
BytesInGetMessage	Size of bsp_get message
BytesInSyncMessage	Size of synhronization message
BSMPMessageMarkSize	Extra message overhead with BSMP
TimeToStartPacketizing	Fixed overhead of comm. controller
TimeToPacketizeOneByte	Overhead pr. byte of comm. controller
ProcessorBus_Width	Width in bits of shared bus
ProcessorBus_Frequency	Speed of shared bus
ProcessorBus_ArbitrationGap	Time used for bus arbitration
Bus_BarrierTreeFanIn	No. of children in sync. reduction tree
Bus_BarrierTreeFanOut	No. of children in sync. broadcast tree

The DSM architecture performs synchronisation by a two step procedure. In the first step, the processors are organised in a reduction tree and messages are sent from the children towards the root to signalise that a processor or a subset of the processors has entered the synchronisation call. The parameter *Bus_BarrierTreeFanIn* gives the fan-in of this tree. When all processors have done the call — the root node knows this and initiates step 2. This involves notifying all the processors that they may proceed — and is done by sending messages from the root node to the leaf nodes in a broadcast tree. *Bus_BarrierTreeFanOut* specifies the width (fan-out) of this tree. Mellor-Crummey and Scott found that the values 4 (fan-in) and 2 (fan-out) might be optimal for these parameters under certain conditions [11]. A detailed study of performance effects of varying these two parameters has been done by Leikvoll [9].

3.5 Tightly Coupled Multiprocessor (TCM)

The tightly coupled multiprocessor model in BSPlab models a network of computing nodes connected by dedicated communication links. The parameter *Network_Topology* specifies the topology of the multiprocessor and the current options are 2D mesh, 3D mesh, 2D torus and hypercube of any dimension. As table 4 shows, there are parameters for determining the size and dimension of a topology.

Table 4. Tightly Coupled Multiprocessor (TCM) architecture parameters

Parameter name	*Informal description*
Network_Topology	Topology of interconnection network
Network_Xsize,...Ysize,...Zsize	Size of network, X, Y and Z dimension
Network_Dim	Number of dimensions in hypercube
Network_PortModel	Channels between processor and router
Network_OutPortPriority	Priorityalgorithm for outgoing message
Network_TransTime	Communication cost pr. byte
Network_StartUpLat	Communication startup cost
Network_RoutingMethod	Way of sending msg through network

The TCM model assumes that each processor is connected to a *router* that interfaces to the communication links. *Network_PortModel* lets the experimenter choose between having one physical communication channel between the processor and the router, or as many channels as there are external links from and to the router. If there is only one channel there must be some mechanism for choosing which of the outgoing messages should first get the channel. *Network_OutPortPriority* decides this and the options are FIFO or random. FIFO is most fair, but random might be useful for breaking up troublesome access patterns. In addition, the TCM model is using a linear communication cost model for the physical links, given by *Network_StartUpLat* and *Network_TransTime*. There are several strategies for sending messages through a network. The parameter *Network_RoutingMethod* offers a choice between store-and-forward, wormhole routing and two-phase randomised wormhole routing [1].

3.6 Network of Workstations (NOW)

The BSPlab model of network of workstations (NOW) contains 12 parameters and models an Ethernet with 10 or 100 Mbit/s bandwidth. Most of the parameters are from the Ethernet standard and the model is given a detailed treatment including recommended parameter values in the original BSPlab documentation [6]. In addition to the standard, the model is based on the Ethernet model in *Simulation Computer Systems* by MacDougall [12]. For brevity, we will not go into detail here.

The NOW model has two parameters for specifying the “noise” generated by other applications using the same communication cable. When switched on, “competing” messages of random length to random processors are sent as a Poisson process. Dybdahl and Uthus validated the NOW model by doing several tests. One of these was a comparison with Lams analytical model [13] of an Ethernet. This shows the normalised delay as a function of the throughput, and the BSPlab experiments demonstrated performance very close to the analytical model.

4 Status and Example Experiment

During development, BSPlab was tested on the test programs following the BSPlib standard and a few simple BSP applications. Lilleaas did a thorough testing of BSPlab where he defined 25 different BSP architectures and tested these on 18 different test programs. Very few problems were found [14]. Leikvoll used the BSPprobe program developed by Hill [5] to measure the BSP-parameters *s*, *l* and *g* for the BSPlab simple machine. Among his findings was that BSPprobe was able to measure *l* and *g* very accurately on this machine. He also found similar results using the BSP-bench program included in BSPPACK from Bisseling [15].

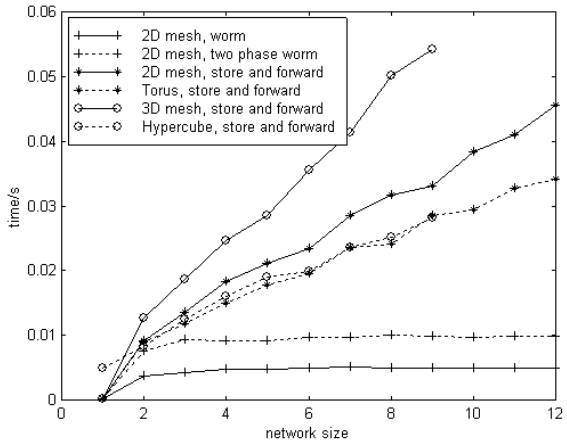

Fig. 1. Performance of 6 different message passing implementations (from [13])

Figure 1 summarises the performance that Lilleaas found running a message-passing test on some of the topologies in the BSPlab TCM model. 2D mesh, 2D torus, 3D mesh and hypercube are tested with wormhole routing and store and forward message passing. The test program does 100 iterations letting a processor send a large message to a random destination and measures the message latency. The time used is plotted as function of the network size with values up to 12. Here the network size is the diameter of the network, which equals the maximum number of hops a message must travel. As expected, the time used by wormhole routing is nearly independent of network size while store and forward has a time consumption that is nearly linear in the number of hops. Further details and many other experiments were documented by Lilleaas [14].

5 Concluding Remarks

Using BSPlab, we have experienced the importance of not varying to many parameters in the same experiment. The interplay between HW and SW in parallel applications is

complicated and a systematic approach starting with simple and understandable experiments is recommended.

Several student projects have been done on testing BSPlab, and we have seen that it is working well. Validation of the performance models in BSPlab is still going on. In this context, we have realised that it is a good tool for demonstrating well known aspects of relations between algorithms, applications and architectures. We are therefore planning to use it as a central part in the teaching of a new course in parallel computer architecture. A step by step text introducing parallel computing aspects and performance effects using BSPlab as demonstration tool is under development. This might also be useful in teaching of BSP programming and parallelism in general. In addition, BSPlab can be used as a research tool in the field of portable and efficient software. Its source code is available from its website and the author appreciates comments on its use and possible improvements.

References

1. Valiant, L. A Bridging Model for Parallel Computation. Communicationsof the ACM, Vol. 33, No. 8 (1990) 103–111
2. McColl, W. F., Bulk Synchronous Parallel Computing, In Abstract Machine Models for Highly Parallel Computers, Davy, J.R. , Dew, P.M. (eds), Oxford Science Publications (1995) 41–63
3. Culler, D., Singh, J.P., Gupta, A.: Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann Publishers, San Fransisco (1999)
4. Hill, J. et al.: BSPlib: The BSP programming library. Parallel Computing, Elsevier Science, Vol. 24 (14) (1998) 1947–1980 (`http://www.bsp-worldwide.org`)
5. Hill J.: BSPprobe. (`www.bsp-worldwide.org/implmnts/oxtool/contrib.html`)
6. Dybdahl, H., Uthus, I.: Simulation of the BSP Model on Different Computer Architectures, Diploma Thesis, Dept. of Computer and Information Science (IDI), NTNU, Trondheim, Norway (1997). (Available from `http://www.idi.ntnu.no/bsplab`)
7. C++SIM User's Guide. Release 1.5. Draft Ver. 1.0, Computing Laboratory, Univ. of Newcastle upon Tyne, UK (1994) (Available from `http://cxxsim.ncl.ac.uk/`)
8. McKinley, P. K., Trefftz, C.: MultiSim: A Tool for the Study of Large-Scale Multiprocessors. Proc. of Int'l. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Networks (MASCOTS), San Diego (1993) 57–62
9. Leikvoll, I.: BSPlab and BSP-parameters. Diploma thesis, Dept. of Computer and Information Science (IDI), NTNU, Trondheim, Norway (1999)
10. Line, S.: Using BSPlab as a PRAM simulator. Diploma thesis, Dept. of Computer and Information Science (IDI), NTNU, Trondheim, Norway (2000)
11. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for Scalable Synchronisation on Shared-Memory Multiprocessors. ACM Trans. on Computer Systems, Vol. 9, No. 1 (1991) 21–65
12. MacDougall: Simulating Computer Systems. MIT Press (1987)
l3. Lam, S.S. A Carrier Sense Multiple Access Protocol for Local Networks. Computer Networks 4 (1980) 21–32
14. Lilleaas, E.: Evaluation of BSPlab. Project Report, Dept. of Computer and Information Science (IDI), NTNU, Trondheim, Norway (1998)
15. Bisseling, R.: BSPpack. From `www.math.uu.nl/people/bisselin/software.html`