COMPUTATIONAL MODELS FOR PARALLEL COMPUTING AND BSPLAB

Lasse Natvig
Group for Computer Architecture and Design
IDI, NTNU, Trondheim, Norway
19 January 1998.

1. INTRODUCTION

A major challenge for parallel computing to become a widespread discipline is the development of a *standardized* combination of *portable and efficient parallel programming*. An interesting approach towards this major goal is the research with offspring in Leslie Valiant's *Bulk Synchronous Parallel Model* (BSP) [Vali90]. The BSP model is a theoretical framework outlining how parallel computations can be organized in a way that bridges the gap between the needs of the programmers and the technological possibilities of the computer designers.

The *BSPlab* project is an attempt to contribute in the BSP research. BSPlab is an environment that makes it possible to develop, debug and measure BSP programs on a variety of parallel computer architectures. It might be a useful tool for increasing the understanding of "what makes a good BSP computer?". The typical use of BSPlab will be experiments to learn about the interplay between hardware and software in the search for efficient and portable parallel programming. The project is an offspring from studies on computational models and an interest in using results from theoretical computer science in practical computer engineering.

The paper starts by motivating studies of theoretical computer science in general and abstract machines (computational models) in particular. Section 1 and 2 is a revised and condensed version of "background material" in "*Evaluating Parallel Algorithms: Theoretical and Practical Aspects* [Natv90a]. The goal of that work was to make a small contribution in bridging the gap between theory and practice in parallel computing. Sections 1 and 2 describe how studies of parallel algorithms led into an interest in abstract machines, the PRAM model and Valiant's BSP model. Section 3 describes Valiant's BSP model, and BSP programming is discussed in Section 4. The paper ends by giving a short description of the BSPlab environment, its goals and the current status.

1.1 MASSIVELY PARALLEL COMPUTING

> *''There are two main communities of parallel algorithm designers: the theoreticians and the practitioners. Theoreticians have been developing algorithms with narrow theory and little practical importance; in contrast, practitioners have been developing algorithms with little theory and narrow practical importance.''* Clyde P. Kruskal in *Efficient Parallel Algorithms: Theory and Practice* [Krus89].

There is no longer any doubt that parallel processing will be used extensively in the future to build the at any time, most powerful general purpose computers. Parallelism has been used for many years in many different ways to increase the speed of computations. Bit-parallel arithmetic (the 1950'ies), multiple functional units (1960'ies), pipelined functional units (1970'ies) and multiprocessing (1980'ies) are all important techniques in this context.

Today, these and many other ways of parallelism are being exploited by a vast number of different computer architectures —- in commercial products and research prototypes. One can not predict in detail which computer architectures will be the dominating for use in the most powerful (parallel) general purpose computers in the future. In the 1980'ies the supercomputer market has been dominated by the pipelined computers (also called vector computers) such as those manufactured by Cray Research Inc., USA, and similar computers from the Japanese companies Fujitsu, Hitachi and Nec. However, the next ten years may change the picture.

At the SUPERCOMPUTING'90 conference in NewYork, November 1990, it seemed to be a widespread opinion among supercomputer users, computer scientists and computational scientists that *massively parallel computers*[1] will outperform the pipelined computers and become the dominating architectural main principle (for supercomputers) in year 2000 or earlier. The main argument for this belief is that the computing power (speed) of massively parallel computers has grown faster, and is expected to continue to grow faster than the speed of pipelined computers.

The inertia caused by the existing supercomputer installations and software applications based on pipelined computers will result in a transitional phase from the time when massively parallel computers are generally regarded to be more powerful to the time when they are dominating the supercomputer market. The actual time where

[1] Systems with over 1000 processors has been the most widely used meaning of this term.

massively parallel computers typically started to outperform the vector machines is not possible to define in a precise and agreed upon manner. Danny Hillis, co-founder of Thinking Machines Corporation and well known for the CM-2 and CM-5 supercomputers, stated in 1990 that the transition (in computing power) happened in 1988 [Hill90]. Other researchers would argue that the vector machines still in 1998 are faster for general purpose processing. However, the important thing is that it seems to be almost a consensus that the transition has taken place or will take place in the near future.

Consequences for research in parallel algorithms

The continuing increase in computing power and the adoption of massively parallel computing have at least two important implications for research and education in the field of parallel algorithms:

- *We will be able to solve larger problems.* Larger problem instances increase the relevance of asymptotical analysis and complexity theory.
- *We will be able to use algorithms requiring a substantially larger number of processors.* One of the characteristics of parallel algorithms developed in theoretical computer science is that they typically require a large number of processors. These algorithms will become more relevant in the future.

To conclude, the possible proliferation of massively parallel systems will make the research done on parallel computations within theoretical computer science more important to the practitioners in the future.

1.2 THE GAP BETWEEN THEORY AND PRACTICE WITHIN PARALLEL COMPUTING

Two worlds of parallel computations

There are at least two main directions of research in parallel computations. In theoretical computer science parallel algorithms are typically described in a high-level mathematical notation for abstract machines, and their performance are typically analysed by assuming infinitely large problems. On the other side we have more practically oriented research, where implemented algorithms are tested and measured on existing computers and finite problems. Both approaches have several advantages. The main differences between these two approaches are also reflected in the literature.

Can the theory be used in practice?

Studies of parallel algorithms described within theoretical computer science (TCS) and textbooks in parallel algorithms and architectures with a more practical (engineering like) approach showed that these two communities only to a little extent referred each others work. This observation led to the following question:

> *Are the theoretically fast parallel algorithms simply not known by the practitioners, is the case that they are not understood, or are they in general judged as without practical value?*

The question led to a curiosity about the possible practical use of parallel algorithms and other results from theoretical computer science. Attempts was done to learn about the border between theoretical aspects *with* and *without* practical use. The work was further encouraged by the NSF — ARC Workshop on Opportunities and Constraints of Parallel Computing [Sanz89], where several researchers argued that bridging the gap between theory and practice in parallel processing should be one of the main goals for future research.

Computations within theoretical computer science are described for *abstract machines* (computational models). The approach chosen to evaluate theoretical algorithms was to implement these on a simulator of the most widely used abstract machine, the PRAM model. The work is documented in [Natv90a] and more compactly in [Natv90b]. The kind of synchronous MIMD programming that was used is described in [Natv94] and [Natv95]. More recently, the research has shifted to the Bulk Synchronous Model (BSP) proposed by Leslie Valiant [Vali90].

2. Models of Parallel Computation

A computing model is an idealised, often mathematical description of an existing or hypothetical computer. In the latter case, it is often called an abstract machine. In this section, we consider the general benefits of using an abstract machine as a concept.

2.1 ADVANTAGES OF USING ABSTRACT MACHINES (COMPUTATIONAL MODELS)

- *Abstractions simplify.* When developing software for any piece of machinery a computing model should make it possible to concentrate on the most important aspects of the hardware, and hiding low-level details.

- *Common platform.* A computing model should be an agreed upon standard among programmers in a project team. This makes it easier to describe and discuss system behaviour and performance of various parts of a computer system. A good model will increase software portability, and also provide a common language for research and teaching.

- *Bridge between software and hardware.* Leslie Valiant [Vali90] attributes the success of the von Neumann model of sequential computations to the fact that it has been a bridge between software and hardware. On the one side the software designers have been producing a diverse world of increasingly useful and resource demanding software assuming this model. On the other side, the hardware designers have been able to exploit new technology in realising more and more powerful computers providing this model. Valiant claims that a similar standardised bridging model for parallel computation is required before general purpose parallel computation can succeed, and he has proposed the BSP (Bulk Synchronous Parallel) model as such.

- *Analysis and performance models.* Computational models have played a crucial role by providing a common base for algorithm analysis and comparison. The RAM model has been, and still is, fundamental for analysis and comparison of sequential algorithms. For parallel computations the P-RAM model of Fortune and Wyllie has made it possible, for a typical analysis of parallel algorithms, to concentrate on a small set of well defined quantities; number of parallel steps (time), number of processors (parallelism) and sometimes also global memory consumption (space). More realistic models typically use a larger number of parameters to describe the machine. On the other side of the spectre we have special purpose performance models which are highly machine dependent and often also specific to a particular algorithm.

2.2 GENERAL PROPERTIES OF PARALLEL COMPUTATIONAL MODELS

What is the "right" model? The selection of the appropriate model for parallel computation is certainly one of the most widely discussed issues among researchers in parallel processing. Some examples of *difficult trade-offs* in this context are easy programming vs. efficient execution (e.g. shared memory vs. message passing), general purpose model vs. special purpose model (e.g. P-RAM vs. machine or algorithm specific models), easy to use as a programming model vs. mathematical convenience (e.g. P-RAM vs. "Boolean circuit families"), and easy to analyse vs. easy to build (e.g. synchronous vs. asynchronous).

Below we describe important properties of a good model. Some of these properties are overlapping, and unfortunately, several of the desired properties are conflicting.

1. *Easy to understand.* Models that are difficult to understand, or complex in some sense (for instance by containing a lot of parameters or allowing several variants) will be less suitable as a common platform. Different ways of understanding the model will lead to different use and reduced possibilities of sharing knowledge. The importance of this issue is exemplified by the PRAM model. In spite of being one of the simplest models for parallel computations it has been described as a SIMD model in several recent textbooks on parallel algorithms. This is wrong.

2. *Easy to use.* Designing parallel algorithms is in general a difficult task. A good model should help the programmer to forget unnecessary low-level details and peculiarities of the underlying machine. The programmer should concentrate on the problem and the possible solution strategies, the model should not add to the

difficulties of the program design process. Simple, high level models are in general most easy to use. They are well suited for teaching and documentation of parallel algorithms. However, when designing software for contemporary parallel computers, one is often forced to use more complicated and detailed models to avoid "loosing contact with the realities". Synchronous models are in general easier to program than asynchronous models. This is reflected by the fact that some authors use the term chaotic models to denote asynchronous models [Gibb89]. Shared memory models seem to be generally more convenient for constructing algorithms.

3. *Well defined.* A good model should be described in a complete and unambiguous way. This is essential for acting as a common platform.

4. *General.* A model is general if it reflects many existing machines or more detailed models. The use of general models yields more portable results. The RAM model has been a very successful general model for uni-processor machines.

5. *Performance representative.* Marc Snir [Snir89] has written an interesting paper discussing at a general level to what extent high level models lead to the creation of programs that are efficient on the real machine. Informally, good models should give a performance rating of (theoretical) algorithms (i.e. run on the model) that is closely related to the rating obtained by running the algorithms on a real computer. In other words, the practically good algorithms should be obtained by refining the theoretically good ones.

6. *Realistic.* Many researchers stress the importance of a computing model to be feasible. Models that can be realised with current technology without violating too many of the model assumptions have many advantages. Above all, they give a model performance which is representative of real performance as discussed above. With respect to feasibility, there is a general agreement that models based on a fixed connection network of processors are easier to realise in a scaleable manner than models based on a global or shared memory [Hwan93]. Unfortunately, the fixed connection models are in general regarded as more difficult to program. Similarly, asynchronous operation of the processors is realistic but generally accepted as leading to more difficult programming.

7. *Durable.* Uzi Vishkin [Vish89] argues that computing models should be robust in time in contrast to technological feasibility which rapidly keeps advancing. Again, the RAM model is an example. Changing models too often will greatly reduce the possibilities of sharing information and building on other work. However, machines will and should change, and new technological opportunities continue to appear. In practice, this is an argument against too realistic models. A similar view has been expressed by Leslie Valiant [Vali90].

8. *Mathematical convenience.* Stephen Cook [Cook85] describes shared memory models as unappealing for an enduring mathematical theory due to the arbitrariness in its detailed definition. He advocates uniform Boolean circuit families as more attractive for such a theory. In my view, models based on circuit families are not suited for expressing large, practical parallel systems. The main problem is that they describe a system at a very low level, and thus gives little help in mastering the complexity of large systems. Most algorithm designers use the PRAM shared memory model, and it should be noted that it contains two drastic assumptions which are introduced for mathematical convenience. Assuming synchronous operation of all the processors makes the notion of "running time" well defined and is crucial for analysing time complexity of algorithms. Assuming unbounded parallelism (i.e. an unbounded number of processors is available) makes it possible to handle asymptotical complexity.

2.3 THE PRAM MODEL AND ITS LINK TO THE BSP MODEL

The CREW (Concurrent Read Exclusive Write) variant of the PRAM model has been the most widely used model for describing parallel algorithms within theoretical computer science. Its leading role may be explained by its good score with respect to the 8 desirable properties given in the previous section. This is summarised in Table 1.

	Property	**Judgement**
1	Easy to understand	Very good, due to its simplicity.
2	Easy to use	Very good, due to simplicity and synchronous behaviour.
3	Well defined	Very good for EREW and CREW PRAM, too many variants of the CRCW PRAM model.
4	General	Very good as a general framework for teaching and sharing knowledge about parallelism.
5	Performance representative	Currently mediocre, but may become good when algorithm development is done in a programming environment that supports performance analysis through simulation.
6	Realistic	Bad, *but acceptable* due to the «technology-distance-argument».
7	Durable	Highly durable as a programming model due to simplicity and the «technology-distance-argument». (Has been used in 20 years for describing parallel algorithms).
8	Mathematical convenience	Acceptable. Much more convenient than many of the recent more realistic models due to the assumptions of synchronous behaviour and unbounded number of processors.

Table 1: A judgement of the PRAM model as a common standard for expressing parallel computations.

In our earlier work on the practical value of theoretical parallel algorithms we gained experience with implementing and debugging parallel algorithms on the PRAM model. One of the things we experienced was that *synchronous MIMD programming* has a lot of benefits — e.g. easy programming, debugging and analysis [Natv95]. Our interest for Valiant's Bulk Synchronous Parallel (BSP) model was stimulated by Valiant's claim found in [Vali90 p. 104];

“*While a PRAM language would be ideal, other styles also may be appropriate.*“

The BSP model is described in Section 3. It was proposed by Valiant as a so-called *bridging model* between hardware and software for parallel computing.

2.3 BRIDGING MODELS IN GENERAL

Leslie Valiant advocates [Vali90] the need for a bridging model for parallel computation, and proposes the BSP model as a candidate for this role. He attributes the success of the von Neumann model of sequential computation to the fact that it has been a bridge between software and hardware. On the one side the software designers have been producing a diverse world of increasingly useful and resource demanding software assuming this model. On the other side the hardware designers have been able to exploit new technology in realising more and more powerful computers providing this model.

Valiant claims that a similar standardised *bridging model* for parallel computation is required before general purpose parallel computation can succeed. It must imply convenient programming so that the software people can accept the model over a long time. Simultaneously, it must be sufficiently powerful for the hardware people to continuously provide better implementations of the model.

A detailed and realistic model will not act as a bridging model because technological improvements are likely to make it old-fashioned too early. For the same reason a bridging model should also be simple and *not* defined at a too detailed level. There should be open design choices allowing better implementations without violating the model assumptions. Valiant’s BSP model can be regarded as a framework. In my opinion this is exemplified by the fact that the BSP model used at Oxford (see Section 3.2) includes some pragmatic design choices necessary for making it into a programming model for execution of real programs.

An example of a bridging model from another field of computer science is the success of the relational model in the database world. This may be explained by the fact that the relational model has acted as a bridge between users of database systems and implementers. When proposed, the relational model was simple and general, high level and "advanced". Database system designers have worked hard for a long time to be able to provide efficient implementations of the relational model.

3. VALIANT'S BULK SYNCHRONOUS PARALLEL (BSP) MODEL

3.1 VALIANT'S ORIGINAL BSP MODEL

Valiant's BSP model of parallel computation is defined as the combination of three attributes: a number of components, a message router and a synchronization facility [Vali90].

- The *components* perform processing and/or memory functions.
- The *router* delivers messages point to point between pairs of components.
- The *synchronization facility* is able to synchronise all or a subset of the components at regular intervals.

The length of the synchronization interval is called the *periodicity* parameter (L), and it may be controlled by the program. A computation is described as a sequence of supersteps. In each *superstep*, each component is allocated a task consisting of some combination of local computation steps, message transmissions and (implicitly) message arrivals from other components. During a superstep the (processing) components perform computations asynchronously. The synchronization facility is used to ensure that all components have finished the current superstep before they proceed to the next[2]. In this sense, the BSP model may be seen as a compromise between asynchronous and synchronous operation.

Valiant states that the components have been separated from the router to emphasise that the tasks of computation and communication *can* be separated. To be able to act as a durable common platform, the model should not specify technical solutions that are irrelevant for the programmer. The programmer needs the functionality of communication, but *increased knowledge of how the communication is implemented will most likely reduce the portability of the software written for the model*. On the other side, the hardware designer should feel free to design clever solutions for offering rapid communication. In the same spirit, there are several architectural choices for how to distribute computation and memory resources on a set of communicating components. Assuming a local memory attached to every processor as done in the Oxford BSP project is one very natural solution, but Valiant's original BSP model allows for other possibilities.

3.2 DESIGN CHOICES IN THE OXFORD BSP MODEL (AND IN BSPLAB)

In his paper on the Oxford BSP library [Mill94], Richard Miller states "The semantics of the Oxford BSP library is based on a slightly simplified version of the model presented in [Vali90]." My interpretation of the papers by the Oxford BSP researchers is that they have settled down some of the design choices that deliberately were left open by Valiant. I guess this has been necessary for making BSP-programming practical. The purpose of this section is to discuss the differences between the Oxford BSP model and the original BSP model proposed by Valiant.

In the paper *Bulk Synchronous Parallel Computing* [McCo95a] William McColl defines a *superstep* as:

> "A sequence of steps, followed by a barrier synchronization at which point any memory accesses take effect. During a superstep, each processor has a set of programs or threads which it has to carry out, and it can do the following:
>
> - perform a number of computation steps, from its set of threads, on values held locally at the start of the superstep
> - send and receive a number of messages corresponding to non-local read and write requests"

[2] A simple way of doing this is to use a barrier synchronization. However, Valiant uses a slightly "looser" specification.

One synchronization call ends a superstep

Valiant [Vali90] describes that synchronization is done every *L* time units. If the current superstep has not finished after this synchronization, the superstep is allocated a new period of *L* time units, and so on. This might be called *periodic synchronization* attempts. The Oxford-BSP model assumes *one* synchronization call at the end of the superstep that will force all processors to wait until it has completed. This is illustrated in Figure 1.

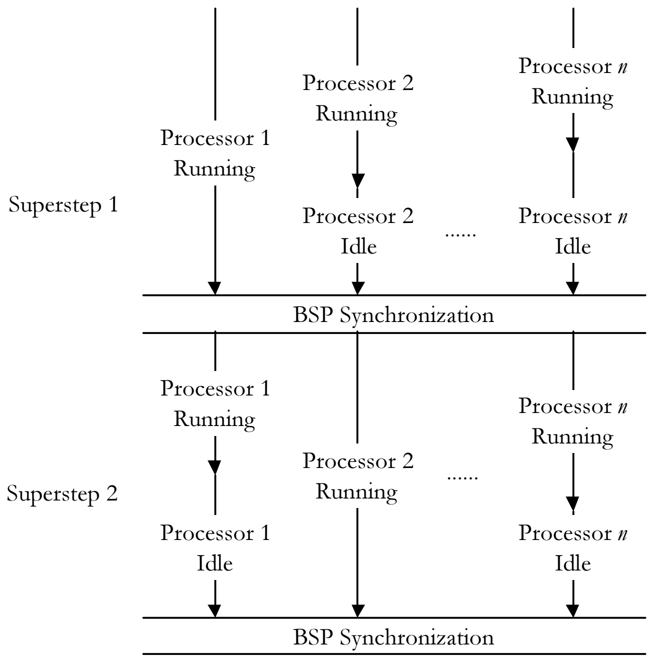

Figure 1: A BSP computation organised in supersteps. (From [DU97]).

The figure shows *n* processors running at the same time. When all processors have signalled that they are finished with a superstep, the BSP synchronization takes place. After the synchronization, all processors continue on the next superstep. In the first superstep, all processors must wait for processor 1 because it finishes last. In the next superstep, processor 2 is the slowest one.

The “one synchronization call method” seems to be the alternative outlined by Valiant [Vali90 p. 105]: *“For example, the system could continuously check whether the current superstep is completed, and allow it to proceed to the next superstep as soon as completion is detected”*. The difference between the two methods are invisible for the programmer; In Valiant’s model the components are not allowed to start on the succeeding superstep before the (last of the periodic) synchronization attempts confirms that all the components have finished the current superstep. Further, Valiant states that if “a minimum amount of *L* time units for this check is charged, the results of the run-time analysis will not change by more than small constant factors”. This lower bound on the time used for synchronization is reflected in the definition of the *periodicity parameter l* used in the Oxford BSP-project, see Table 2 in Section 3.3 below.

The current version of the BSP simulator in BSPlab is using the “one synchronization call method”. In later versions, alternative HW/SW solutions may be investigated for implementation of the synchronization task, since it is known to be crucial for performance. In that context, the original specification of Valiant should be a starting point.

Interference-free supersteps

The Oxford BSP project has also specified in more detail the semantics of access to non-local memory, *i.e.* communication steps. Valiant's definition of a superstep implies that the only guarantee that can be given is that global memory effects are visible at the start of the next superstep. So far the Oxford-BSP and Valiant-BSP agree. From Valiant's definition it is *possible* that the effect of a communication step becomes visible in the same superstep as it is initiated. This may give the possibility of direct, fast communication between pairs of components within the same superstep. It may increase performance, but introduces unpredictable behaviour since the computation becomes dependent on timing in the various components. This does not seem to be in "the spirit of bulk-synchronous behaviour", since an important motivation for inserting synchronization points in a computation is the advantage of being able to assume a well-defined state at given points in the program.

The definition of a superstep given by McColl [McCo95a] (see above) simplifies the situation by stating that the memory accesses take effect at the end of a superstep. A consequence is that it can be guaranteed that their effects are *not* visible within the same superstep. In addition to simplifying the semantics of a BSP program, this gives several opportunities for performance enhancements in a BSP computer. Several smaller communication operations may be joined together in one larger operation to reduce overhead, or they may be scheduled in a particular order which is assumed to give faster total delivery. Another possibility is to combine the task of communication and synchronization, e.g. by letting synchronization messages "piggyback" on communication messages.

In the Oxford BSP library, the simple semantics of supersteps as defined by McColl is realized by requiring that the BSP programs adhere to a set of *non-interference rules*. Informally these can be stated as "within each superstep, locally manipulated data must be disjoint from remotely communicated data". This ensures that the semantics of a superstep becomes independent of when remote communications are completed within a superstep. Thus, the definition by McColl is included as one possibility. Further details can be found in The Oxford BSP Library User's Guide [MR95]. The current version of BSPlab performs all remote memory accesses at the end of a superstep. Later versions will allow experimenting with various implementation alternatives as mentioned above.

Components and synchronization

Both Valiant and McColl write that the periodic synchronization involves all or a subset of the components (processors). However, in [Mill94 p. 2] the barrier synchronization is specified to include all the processors. The same simplifying choice is done in the current version of BSPlab, but the possibility of switching off the synchronization facility for a subset of the processors as described by Valiant will be included in later versions. This facility may be omitted when considering one single BSP program running alone on a BSP-computer. In a more complete setting, e.g. several BSP applications running in a timeshared-fashion on top of a BSP operating system, the possibility of selective synchronization will most likely be a necessary feature.

Valiant describes that the components may have processing and/or memory functions, while the Oxford BSP papers describe that the processors has local memory. The first version of BSPlab adopts the Oxford model.

3.3 MAIN PARAMETERS OF A BSP COMPUTER

The performance of a computer based on the BSP model can be expressed by the four global parameters p, s, l and g The meaning of these are summarised in the Table 2. The definitions follow the Oxford version of the BSP model [McCo95a].

Table 2:_Central parameters of a BSP computer.

	Name	Explanation
p	number of processors	
s	processor speed	number of local operations performed by a processor per second
l	synchronization periodicity	minimal number of time steps between successive synchronization operations
g	global computation/communication balance	(total number of local operations performed by all processors in one second) / (total number of words delivered by the communications network in one second)

A *time step*, as used in the definition of *l*, is the time required for a single local operation, i.e. a basic processor operation on values stored in registers or the local memory of that processor. For a given kind of processor, the MIPS rate should give a good estimate for the length of a time step and thus also the processor speed. For scientific applications it is common to take one *flop* as a (basic) time step [MR95].

Global operations are those that involve access to data not stored local to a processor. The time used to process such remote accesses should (as seen by the programmer) be independent of its location, *i.e.* a BSP computer has a two-level memory model. This choice is crucial for making BSP programs portable since an increased visibility of the memory hierarchy most likely will be exploited in architecture dependent tuning of high-performance applications.

Note that the parameter *g* is defined as a *global* measure as the ratio between the total processing power and the total communication power[3]. As stated by McColl [McCo95a], it is an important feature of the BSP model that the parameters *l* and *g* characterise the communication performance of a BSP computer on a much higher level than what is common. We are normally being told about the speed of individual communication channels, topology and other details which tempts the programmer to exploit locality to increase performance. However, the BSP model helps the programmer to produce portable code by concentrating on the overall performance characteristics of the BSP computer. The high abstraction level and simplicity of the general BSP parameters make it possible to use them as parameters in BSP algorithms. In this way the often conflicting goals of high performance and portability may be combined. (See Section 4.4 on parameterised algorithms.)

4. BSP PROGRAMMING AND BSPLIB

This section starts by describing two general properties of BSP programs that are important for efficient execution on BSP computers. Then we give a short summary of the BSP library developed at Oxford Parallel — the Parallel Computing Center of Oxford University. This is followed by a short outline of BSPlib, the standard library for BSP programming proposed by the BSP WorldWide organisation. The last subsection describes how the BSP parameters can be included in BSP algorithms to achieve both portability and efficient execution. The section ends by suggesting a few references for further studies in BSP programming.

4.1 PARALLEL SLACKNESS AND COMMUNICATION SLACKNESS

BSP programs should have so-called *parallel slackness*, which means that they are written for *v* virtual processors to run on *p* physical processors where *v* is rather larger than *p* (e.g. $v = p \log p$) [Vali90]. This is necessary for the compiler and the run-time system to be able to "massage" and assemble bulks of instructions executed as supersteps

[3] The parameter *g* is very similar to the concept of *machine granularity* as described by Gordon Bell [Bell92 p. 34]. Bell argues that the *problem granularity* should be greater than the machine granularity for a parallel computer to operate efficiently.

giving an overall efficient execution. Parallel slackness[4] is an important concept for parallel computing in general, not only in the BSP context. Programs with larger parallelism than what is available on the machine can, as an example, be used to hide the latency of slow memory accesses as in multithreading [McCo95a].

Communication slackness is necessary for efficient execution of programs on BSP computers with a high value of the parameter *g*. Such computers have high computing performance compared to communication performance, and communication slackness means that for every non-local memory access a node should be able to perform approximately *g* operations on local data [McCo95a].

4.2 THE OXFORD BSP LIBRARY

The Oxford BSP library contains two core routines for the management of each of the three main components in the BSP model. Processors are allocated by `bsp_start` and deallocated by `bsp_finish`, communication is handled by `bsp_fetch` and `bsp_store`, and synchronization is implicitly given by the supersteps which are defined by `bsp_sstep` and `bsp_sstep_end`. The semantics of these six routines are described by Miller and Reed in [Mill94, MR95], both papers easily accessible from WWW [Ravi95].

The Oxford BSP library is a simple platform for writing BSP programs in standard sequential languages such as FORTRAN, C or Pascal. The library routines are architecture independent and available on a large range of architectures. A few high-level templates (`bsp_broadcast` and `bsp_reduce`) have also been provided for convenience and efficiency.

Miller emphasises the following six main properties of the Oxford BSP library programming model:

1. *Data parallel program structure.* It is expected that programs with sufficiently parallel slackness are most likely to occur by decomposing large computational problems by partitioning the data and distributing the load in a data parallel way.

2. *SPMD program text.* A data parallel control structure is easily expressed in a *single program multiple data* programming style.

3. *Direct mode of global memory management.* The programmer determines the partitioning of data among the processors and also explicitly controls the exchange of data from one local memory to another[5].

4. *Remote assignment semantics for data access between processors.* These assignments are asynchronous and the semantics is strongly linked to the notion of a superstep. Shared access to remote data may give non-deterministic behaviour unless the programmer obeys the non-interference rules mention above.

5. *Static processor allocation.* The number of processors used in a computation is allocated at the beginning of a program.

6. *Single threaded process management.* The term processor and process are synonyms in programs using the current version of the Oxford BSP library. Since the multiplexing of several processes on one processor is an important way of achieving parallel slackness, it might be expected that this limitation will be removed in future versions of the library.

[4] Also called *overdecomposition.*

[5] The alternative *automatic memory management* is an important part of the BSP concept [Vali90] and is a typical candidate for experiments with BSPlab.

As a curiosity, it can be mentioned that 5 of the above properties (all except no. 4) apply to the kind of synchronous MIMD programming we carried out as part of studies of PRAM algorithms for sorting [Natv90a].

4.3 THE WORLDWIDE BSPLIB STANDARD

The BSP Worldwide organisation developed a standard proposal for a BSP library during 1996. Since the proposal resulted from a discussion among BSP researchers around the world it was not a surprise that it is slightly bigger than the Oxford BSP library. However, having only 21 basic functions it is one of the simplest libraries for parallel programming. The library is implemented in BSPlab. The functions in the BSP Worldwide standard proposal can be grouped into seven categories as shown in Table 3. The *initialisation* functions is for starting and stopping a BSP program, and for allocation of processors. The *inquiry* group makes the number of available processors, the number of currently used processors, the processor-identification and time available to the BSP program issuing these functions. «Direct Remote Memory Access» (*DRMA*) and «Bulk Synchronous Message Passing» (*BSMP*) are two alternative methods for data transport between processors. *The high performance* functions typically use pointer-exchanging instead of copying.

Class	**Operation**
Initialisation	**`bsp_init`**
	`bsp_begin`
	`bsp_end`
Inquiry	**`bsp_nprocs`**
	`bsp_pid`
	`bsp_time`
Synchronization	**`bsp_sync`**
DRMA	**`bsp_pushregister`**
	`bsp_popregister`
	`bsp_get`
	`bsp_put`
BSMP	**`bsp_set_tag_size`**
	`bsp_send`
	`bsp_get_tag`
	`bsp_move`
High performance	**`bsp_hpput`**
	`bsp_hpget`
	`bsp_hpmove`
Halt	**`bsp_abort`**

Table 3: The operations in the BSP worldwide standard library (from [BSP96])

4.4 PORTABLE SOFTWARE THROUGH USE OF PARAMETERISED ALGORITHMS

The *architectural parameters* of a BSP computer play a central role in the development of transportable parallel programs. Transportable BSP software can be achieved by parametrising algorithms with the BSP parameters *g* and *l* in addition to number of processors (*p*) and problem size (*n*) which are common parameters in parallel algorithms [Vali90]. The BSP papers give various example algorithms with complexity analysis for execution on the BSP model. The cost of an algorithm is typically expressed as a function of *p, n, l,* and *g*. For matrix multiplication, Valiant mentions two different algorithms that give optimal runtime for different values of *g* and *l*. The second algorithm requires fewer messages to be exchanged and can achieve optimal runtime for a larger value of *g* (which implies a computer with lesser communication power). Another example is given in [MR95]. Here the cost of broadcasting of an *n*-element array is analysed for a linear algorithm and for a logarithmic (binary tree) algorithm. The logarithmic version is more complex but will be quicker for *n* equal or larger than a given expression in *p, l,* and *g*.

In general, the complexity analysis of BSP algorithms will indicate that different algorithms should be selected for performing the computation for different values of the problem size and the architectural parameters. Both *l* and *g* may vary with the number of processors *p*. For some scaleable multiprocessor architectures each parameter may be expressed as some function *f(p)*, where *f* typically depends on several factors including the network topology used to interconnect the processors [McCo95a]. The parameters *g, l* and *p* form a *"BSP space"* of parallel computers [McCo95b]. Writing algorithms that automatically select the right algorithmic strategy depending on the location in this space and the problem size is a difficult task. It requires BSP complexity analysis combined with the development of the algorithm. Nevertheless, we believe this is a right way to go for developing standardised, portable and efficient parallel code for mathematical and other libraries. It is worthwhile since the alternative is rewriting the code to achieve efficiency for *every* new parallel computer that has significantly new architectural parameters. In the exploration of such algorithms we believe a BSP simulator designed for easy variation of these architectural parameters will be a valuable tool. In addition, realistic modelling of the time consumption in BSP software and hardware may uncover some of the practical effects of the complexity constants hidden by the Big-O-notation in the BSP complexity analysis.

4.3 LANGUAGES FOR BSP PROGRAMMING, FURTHER READING

Before any attempts are made to design a parallel programming language suitable for the BSP programming model, one should study the ongoing projects in the area. Some of those not mentioned elsewhere in this paper are the following:

- **GPL** is an architecture independent language developed at the Oxford University. It is procedural, block-structured and strongly typed. It is developed as part of the ESPRIT project GEPPCOM (Foundations of General Purpose Parallel Computing) [McCo94].

- **BSP++** is a BSP extension of C++ developed by David Lecomber at the Oxford University [Leco94].

- **OPAL** is an imperative language with Ada-like modules developed by Simon Knee also at Oxford [Knee94].

- **BSP-L** is an experimental BSP programming language being developed at Harvard [CF+95]. A very interesting aspect of this work is that the language is being developed together with studies of optimizing compilation and run time systems in the BSP context.

- **FORK** is a programming language designed for the PRAM model [Hage91]. It is part of the Saarbrucken parallel computer PRAM project, see `http://www-wjp.cs.uni-sb.de/SBPRAM/`.

5. BSPLAB — A VIRTUAL LABORATORY FOR BSP ARCHITECTURES

BSPlab is an environment that makes it possible to study parallel applications using the BSP Worldwide library and executing on different parallel computer architectures. The simulator contains performance models for various common architectures, and thus experiments "iterating over a set of architectures" is possible. This explains the name BSPlab, a (virtual) laboratory offering the use of several (simulated) computer architectures.

This section starts by describing the main goals for the BSPlab project. This is followed by a short introduction to the current version of BSPlab.

5.1 MAIN FUNCTIONALITY AND GENERAL REQUIREMENTS

It is the intention to use the BSPlab environment in several directions of research. In the current state of the project an increased understanding of BSP architectures and computations is the main goal, but several other research directions may benefit from BSPlab:

- *Exploration of the "BSP design space"* of parallel computers. BSPlab should offer the possibility of executing BSP programs on different variants of Valiant's BSP model. The user should be able to specify that a BSP-program should run on a range of BSP computers with varying values for the architectural parameters *p*, *l*, and *g*. Simple methods should be provided for describing what parts of the design space that should be explored, and for describing dependencies among the parameters.

- *Exploring of BSP variants and design alternatives.* The main function of BSPlab is the ability of modelling various ways of implementing BSP computers. Various design alternatives for the main parts of a BSP computer should be modelled. Examples are hashing algorithms for automatic memory management, or an evaluation of two-phase random routing compared with other routing algorithms.

- *Research in BSP systems and hardware.* The production of realistic traces of computation, communication, synchronization and memory access in BSP computers will make it possible to initiate various research projects aiming at describing or realising prototypes of hardware and/or software modules for BSP computers. Our research group at NTNU has a special interest in reconfigurable hardware, which today is feasible through the use of FPGA-devices and similar technologies. This concept may be interesting for BSP computers, since it may be implemented as a system alternating between computing and communicating (and synchronizing) phases.

- *Modelling of communication in message passing computers.* Pauline Haddow is currently doing research on performance modelling of the communication in message passing computers [Hadd98]. She is developing equations for the latency involved in message passing on various communication architectures. These equations may be useful in making realistic high-level abstractions of communication solutions for BSP computers. The opposite way, communication traces from BSP applications run on the simulator may be useful as benchmarks for her work.

- *Research in BSP software.* The simulator may be used as an execution tool in studies of BSP programming. Our interest in BSP programming stems from experience with PRAM programming as mentioned earlier in the paper. A general BSP simulator can easily be turned into a PRAM simulator by setting the parameters *l* and *g* equal to 1. PRAM programming might be ideal for teaching purposes and theoretical research in parallel algorithms. Today, a more practical approach for developing transportable and efficient parallel programs for real applications — is the use of BSP computers with realistic values for *l* and *g*, as a cost-effective way of executing PRAM programs. Crucial in this context is compiling and other techniques for transforming PRAM programs into an appropriate sequence of supersteps. Our work on *compile and runtime padding* may be relevant in this context [Natv94].

5.2 A SHORT INTRODUCTION TO THE CURRENT VERSION OF BSPLAB

BSPlab is an environment for experimenting with BSP programs on different computer architectures. Parallel applications are written in Visual C++ from Microsoft using the BSPlib standard for communication and synchronisation. The BSPlib is the result of the standardisation effort by the BSP World Wide organisation (`http://www.bsp-worldwide.org/`).

In BSPlab, the user may select among various predefined computer architectures. Currently, these are multiprocessors with shared memory or distributed shared memory, network of workstations, and multicomputers with message passing organised in various network topologies. The user may also define her own architectures. Some features are common for all the architectures currently provided by BSPlab:

- Components are *processors with local memory* as in the Oxford BSP model.

- The *router* delivers messages point to point between processors. Several HW/SW designs for implementing this communication is offered.

- The *synchronization* facility is only able to synchronise *all* the processors, and one synchronization call ends a superstep.

- The semantics of a *superstep* is as described by McColl [McCo95a] (see Section 3.2.). This implies that access to remote memory take effect at the end of a superstep.
- It is only possible to run one single BSP program at a time.

The BSP programs are debugged and executed in BSPlab to achieve (simulated) performance measures and hopefully a better understanding of the interplay between hardware (i.e. the selected BSP-architecture) and software (i.e. the executed BSP-program). BSPlab is primarily an environment for studying BSP computations with focus on the impact of algorithms and architectures on BSP program performance. However, it can also be used as a programming environment for BSP applications developed for real parallel computers running BSPlib.

BSPlab was developed as the diploma work of Haakon Dybdahl and Ivan Uthus from august 1996 to 1997. A web-site, "`http://www.idi.ntnu.no/bsplab`", has been made to offer the BSPlab to interested researchers world wide. Here you can find the necessary files needed for making your own installation of BSPlab, example programs and documentation (including the Diploma thesis by Dybdahl and Uthus).

REFERENCES

[Bell92] Gordon Bell, ULTRACOMPUTERS, A Teraflop Before Its Time*, Communications of the ACM,* Vol. 35, aug 1992, pages 27-47.

[BSP96] Mark W. Goudreau, Jonathan M. D. Hill, Kevin Lang, Bill McColl, Satish B. Rao, Dan C. Stefanescu, Torsten Suel and Thanasis Santilas, «A Proposal for the BSP WorldWide Standard Library», July 1996. Can be retrieved from http://www.bsp-worldwide.org/

[CF+95] T. Cheatham, A. Fahmy, D. Stefanescu, and L. Valiant, Bulk Synchronous Parallel Computing—- A Paradigm for Transportable Software, In Proceedings of the 28th Annual *Hawaii Conference on System Sciences* volume II, IEEE Computer Society Press, January 1995.

[Cook85] Cook, Stephen A., *A Taxonomy of Problems with Fast Parallel Algorithms*,, Information and Control, vol. 64, pages 2 - 22, 1985.

[DU97] Dybdahl, Haakon and Uthus, Ivan, *Simulation of the BSP Model on Different Computer Architectures*, Diploma Thesis, *Department of Computer and Information Systems (IDI), Norwegian University of Science and Technology (NTNU),Trondheim, Norway. February 1997. Partly available from* "`http://www.idi.ntnu.no/bsplab`".

[Gibb89] Phillip B. Gibbons. Towards Better Shared Memory Programming Models, In *Proceedings of the NSF - ARC Workshop on Opportunities and Constraints of Parallel Computing*, San Jose, California, December 1988, pages 55-58. Springer-Verlag 1989.

[Hadd98] Haddow, Pauline, *A Framework for Modelling Communication Hardware in MIMD Computers (preliminary title),* PhD-thesis, Norwegian University of Science and Technology, 1998.

[Hage91] T. Hagerup, A. Schmitt, H. Seidl. *FORK*, A High-Level Language for PRAMs. In *Proceedings of Parallel Architectures and Languages Europe'91*. pages 304-320. Springer Verlag 19....

[Hill90] Hillis, W. Daniel, *The Fastest Computers*, Keynote Address at the SUPERCOMPUTING'90 conference, New York, 13 November, 1990. Notes from the presentation.

[Hwan93] Hwang, Kai, *Advanced Computer Architecture, Parallelism, Scalability, Programmability*. McGraw Hill, 1993

[Knee94] Simon Knee, Program development and performance prediction on BSP machines using opal, echnical report PRG-TR-18-1994, Oxford University Computing Laboratory, Oxford University 1994.

[Krus89] Kruskal, Clyde P., *Efficient Parallel Algorithms: Theory and Practice*, in Proceedings of the NSF - ARC Workshop on Opportunities and Constraints of Parallel Computing [Sanz89], pages = 77—79, 1989.

[Leco94] David Lecomber. An Object-Oriented Programming Model for BSP Computations, Technical report, Oxford University, Dec. 1994, 32 pages.

[McCo94] W F McColl, BSP Programming, In *Proc. DIMACS Workshop on Specification of Parallel Algorithms*, American Mathematical Society, Princeton, May 94. G Blelloch, M Chandy, and S Jagannathan , editors,

[McCo95a] McColl, William F., Bulk Synchronous Parallel Computing, In *Abstract Machine Models for Highly Parallel Computers*, Editors J. R. Davy and P. M. Dew, Oxford Science Publications, pages 41-63, 1995.

[McCo95b] W. F. McColl, The BSP Approach to Architecture Independent Parallel Programming, Oxford University Computing Laboratory, March 21th, 1995.

[Mill94] Richard Miller, Library for Bulk-synchronous Parallel Programming, In British Computer Society Parallel Processing Specialist Group workshop on General Purpose Parallel Computing, December 1993.

[MR95] Richard Miller, and Joy Reed, The Oxford BSP Library Users' Guide, Oxford Parallel, Oxford University Computing Laboratory, 1.0 edition, May 1994.

[Natv90a] Lasse Natvig. *Evaluating Parallel Algorithms: Theoretical and Practical Aspects*, PhD thesis, Division of Computer Systems and Telematics, The Norwegian Institute of Technology, The University of Trondheim, Norway, Dec. 1990.

[Natv90b] Natvig, Lasse, *Logarithmic Time Cost Optimal Parallel Sorting is Not Yet Fast in Practice!*, Proceedings of SUPERCOMPUTING'90, New York, November 1990, pages 486 -494

[Natv94] Lasse Natvig, Compile and Runtime Padding: An Approach to Realising Synchronous MIMD Execution, *IFIP World Computer Congress, Track 4: Theoretical Foundations of Computing*, IFIP Transactions A-51, Volume I, page 553-558, Hamburg, Aug 1994.

[Natv95] Lasse Natvig. General Purpose Parallel Programming on the PRAM Model. In *Abstract Machine Models for Highly Parallel Computers*, Editors J. R. Davy and P. M. Dew, Oxford Science Publications , pages 66-83, 1995.

[Ravi95] Ravi Palepu, Bulk Synchronous Parallel (BSP) Repository (html), School of Computer Science, Carleton University, `http://www.scs.carleton.ca/~palepu/BSP.html`

[Sanz89] Sanz, Jorge L. C., *Opportunities and Constraints of Parallel Computing*, 1989, Springer-Verlag, London. Papers presented at the NSF - ARC Workshop on Opportunities and Constraints of Parallel Computing, San Jose, California, December 1988. (ARC = IBM Almaden Research Center, NSF = National Science Foundation)

[Snir89] Marc Snir, Parallel Computation Models—-Some Useful Questions. In *Proceedings of the NSF - ARC Workshop on Opportunities and Constraints of Parallel Computing*, San Jose, California, December 1988, pages 139-145. Springer-Verlag 1989.

[Vish89] Uzi Vishkin. PRAM Algorithms: Teach and Preach, In *Proceedings of the NSF - ARC Workshop on Opportunities and Constraints of Parallel Computing*, San Jose, California, December 1988, pages 161-163. Springer-Verlag 1989.

[Vali90] Leslie G. Valiant. A Bridging Model for Parallel Computation, *Communications of the ACM*, Vol. 33, No. 8, August 1990, pages 103-111.