A brief introduction to the BSP model

1 Introduction

Since the introduction and adaptation of the von-Neumann model for sequential computing, the effects of computer revolution on society have been pretty significant. A general purpose computer performs well on computer programs written on a variety of standardized programming languages like C, Fortran, Cobol or Lisp and the same computer program can be easily ported on other platforms.

It has always been realized that parallel computers will eventually supersede sequential machines. This has yet to happen despite advances in computer technology and the fact that chip technology seems to have reached physical limitations; nowadays, fast machines are not much faster than the slowest ones which are as fast (or perhaps faster) as a supercomputer of twenty years ago. Small incremental improvements that may lead to stagnation of sequential computing seem inevitable. Despite these shortcomings of sequential computing, there has been no significant spread of use of parallel computers and few companies have realized that their future may rely on parallel platforms. The main reason for this has been that parallel computer platforms are built in such a way that are too hardware specific, programs written for them exhibit poor performance unless the programmer fine-tunes its code to take into consideration features of the particular architecture. Not only the code is non-portable but scalability comes at a high cost as well. On the other hand parallel algorithms designed and analyzed by the theorists work on parallel models that usually ignore communication and/or synchronization issues, like the PRAM and its variants, and work only under unlimited parallelism assumptions.

One of the earliest attempts to model a parallel computer has been the Parallel Random Access Machine (PRAM) which is one of the most widely studied abstract parallel models. A PRAM consists of a collection of processors which work synchronously and which communicate with a global shared random access memory which can access in unit time. There are many different types of PRAMs which are distinguished from the way they access the shared memory (eg CRCW, EREW PRAMs). Numerous parallel algorithms have been developed for this parallel computer model.

More realistic models of parallel computing view a parallel computer as a collection of sequential processors, each one having its own local memory (*distributed-memory model*). The processors are interconnected by a network which allows them to communicate by sending and receiving messages. Constraints such as the maximum number of pins on a chip, or the maximum width of a data bus, limit the capacity of a processor to communicate with any other processor. It is only possible for a single processor to communicate directly with few others, in most cases those physically close to it. If a message needs to be sent to a distant processor it is relayed through a number of intermediate processors.

As it has already been mentioned, the parallel machines built in the 1980s and early 90s failed to garner general acceptance mainly because of the lack of a stable, unified and bridging parallel programming model. These deficiencies made programming of such machines difficult (cf. assembly vs higher level programming languages), time consuming, non-portable and architecture-specific. Recently, the introduction of realistic parallel computer models such as the Bulk-Synchronous Parallel (BSP) model of computation by L.G. Valiant [9] comes to address these limitations of parallel computing. Our hope is that further architectural convergences will occur (as evident with the proposed Virtual Interface Architecture Specification for System Area Networks proposed by Compaq, Intel and Microsoft and supported by about 100 other companies) with the goal of writing software that will be portable and run with high performance on a variety of architectures from networks/clusters of workstations (NOW/COW) to parallel supercomputers.

1.1 The BSP Model

The *Bulk-Synchronous Parallel* (BSP) model of computation has been proposed by L.G. Valiant [9], as a unified framework for the design, analysis and programming of general purpose parallel computing systems. It allows the design of algorithms that are both scalable and portable. In a BSP program, processors jointly advance through its various phases called *supersteps* with the required computations and remote communication occurring between them; at the end of a superstep processors check themselves in order to proceed to the following superstep.

The BSP model, as described in [9], consists of three parts:

(1) a collection of *processor-memory components*,

(2) a *communication network* that can deliver messages point-to-point among the components, and

(3) a *facility for global synchronization*, in barrier style, of all or a subset of the components.

The model does not assume the existence of any special facilities for broadcasting or combining nor does it deal directly with issues such as input and output, though it can be augmented to take such issues into consideration. A time step (as opposed to a CPU instruction or cycle) would refer to the time needed to perform a local computation (such as a fetch from memory and a floating-point operation followed by a store operation).

It should be noted that, although the model stresses global barrier-style synchronization, pairs of processing units may synchronize pairwise by sending messages to and from an agreed memory location. However, such message exchanges should respect the superstep rules [2]. Computation on the BSP model proceeds in a succession of *supersteps*. A superstep may be thought of as a segment of computation during which each processor performs a given task using data already available there locally before the start of the superstep. Such a task may include (i) local computations, (ii) message transmissions, and (iii) message receipts.

The tuple (p, L, g) characterizes the behavior and performance of a BSP computer. Parameter p is the number of components available, L is the minimum time between successive synchronization operations, and g is the ratio of the total throughput of the whole system (in a steady state, i.e. in a state of continuous message usage) in terms of basic computational operations, to the throughput of the communication network in terms of words of information delivered. A lower bound on the value of L is the time for a remote memory operation/message dispatch to become effective and is thus dependent on the diameter of the interconnection network. The time for barrier synchronization also poses a lower bound on the effective value of L. The value of g is measured while the network is in a steady state, i.e. latency issues become insignificant in the measurement of communication time; parameter L is large enough for the theoretical bound on g to be realizable. An upper bound on L is application specific and expressed in terms of problem size n as well. Message size is ignored in the BSP model by being abstracted away.

The theoretical definition of g relates to the routing of h-relations; when each processor sends or receives at most h messages (of apparently, the same size) an h-relation is realized and the cost assigned to this communication is gh provided that $h \geq h_0$, where h_0 is a machine dependent parameter. Otherwise the cost of communication is L. This way latency issues associated with small message size/messages are taken into consideration. In practice, and in various libraries implementing the BSP model message requests issued/accepted by processors are of variable size. The h of an h-relation relates then to the total size of communicated data and g is expressed in terms of basic computational operations (sometimes, floating-point operations) or absolute time (seconds) per data-unit (a byte or word of information). The BSP model has been augmented to take into consideration a block parameter B; in the resulting BSP* model, if an h-relation is routed with each message of maximum message size s then a cost of $h \lceil s/B \rceil g$ is assigned to this communication or L, whichever is largest. In practice, the parameter L of the BSP model not only hides the cost of communication when each processor sends or receives a small number of messages but also the cost of communication where each processor may send or receive a large number of messages but each one is of small size. For any BSP computer, the values of L and g are likely to be a non-decreasing functions of p.

The use of L and g to characterize the communication and synchronization performance of a BSP computer is important because such issues are abstracted in only two scalar parameters thus allowing considerations to be moved from a local level to a global one.

For the sake of an example, the values for L and g for some abstract network topologies are as follows. A ring has $L = O(p), g = O(p)$, a 2d-array (mesh) $L = O(\sqrt{p}), g = O(\sqrt{p})$, a butterfly $L = O(\log p), g = O(\log p)$, and a hypercube $L = O(\log p), g = O(1)$. For the case of a hypercube, as $g = O(1)$ the cost of routing a permutation, i.e. an one-relation is not $1 \cdot g$ but L. Thus h_0 for the hypercube is such that $h_0 = \Theta(\log p)$.

Taking into consideration the previous discussion, a cost of $\max\{L, x + gh\}$ basic time steps is assigned to a superstep S, where x is the maximum number of basic computational operations executed by any processor during S, and h is the maximum number of messages sent or received by any processor. If h_s is the maximum number of messages sent and h_r is the maximum number of messages received by any processor during S, then $h = \max\{h_s, h_r\}$. One could also have taken $h = h_s + h_r$. Alternative costs are $\max\{L, x, gh\}$ [9] and $L + x + gh$. The maximum size of a superstep depends on the problem in hand. Under the BSP programming paradigm the objective is to maximize the size of supersteps, decrease their number and increase processor utilization.

The description of BSP programs can be simplified by separating computation and communication [2] and assuming that each superstep contains either local computations or communication.

2 Optimality and Analysis of BSP Algorithms

Two modes of programming on the BSP model were envisaged in [9]: *automatic mode* where programs are written, say PRAM style, in a high level language that hides memory distribution from the user and *direct mode* where the programmer retains control of memory allocation. In the *direct mode* of programming small multiplicative constant factors in runtime are important. It can be shown that the automatic mode achieves optimality within constant factors by simulating say PRAM algorithms on the BSP [2]. The term *slack* in the context of algorithm design refers to the ratio of the problem size over the processor number of the BSP machine. The question is whether the direct mode can be beneficial in circumstances where:

- small multiplicative constant factors in runtime are important,
- where smaller problem instances can be run more efficiently in direct mode (less slack is required) than in automatic mode,
- where the available machine has high g (automatic mode requires g to be constant), and
- L is high for direct but not for automatic mode for the problem instance in hand.

Although it is difficult to measure performance to high accuracy, since the operations that are counted have to be carefully defined, we can make such measures meaningful by measuring ratios between runtimes on pairs of models that have the same set of local instructions.

The performance of a BSP algorithm P is thus described in three parts. An algorithm S (say, a sequential one) with which we are comparing S is first specified. The model of computation used for both algorithms is then defined and the basic computational operations that will be counted in both P and S are also described and the charging policy is made explicit. Second, two ratios π and μ are specified. The former, π, is the ratio between the computation time C_P, of the BSP algorithm, over the time C_S of the comparing sequential algorithm divided by p, i.e., $\pi = pC_P/C_S$; the denominator of π is the best achievable parallel time of a parallelization of S with optimal speedup p. The latter, μ, is the ratio between the communication time M_P required by the communication supersteps of the BSP algorithm and the computation time of S divided by p, i.e., $\mu = pM_P/C_S$. When communication time is described, it is necessary that the amount of information that can be conveyed in a single message be made explicit. Finally, conditions on n, p, L and g are specified that are sufficient for the algorithm to be plausible and the claimed bounds on π and μ to be valid. Corollaries describe sufficient conditions for the most interesting optimality criteria, such as c-optimality, i.e., $\pi = c + o(1)$ and $\mu = o(1)$. All asymptotic bounds refer to the problem size as $n \to \infty$.

The mechanisms of the BSP model are similar to those of a sequential one; as performance ratios between them are expressed in terms of π and μ, operations may be defined in a higher level of abstraction than machine level instructions.

3 Software Support under the BSP model

The BSP model, unlike other models of parallel computation, is not just an architectural-oriented theoretical model; it can also serve as a paradigm for programming parallel computers. The fundamental concept introduced by the BSP model is the notion of the superstep, and that all remote memory accesses occur between supersteps as part of a global operation among the processors; the results of these accesses become effective at the end of the current superstep. Although it may have been apparent why some consider the BSP as a satisfactory unifying and bridging model for parallel computation, one may ask the question of how successful it has been as a practical model and what level of software support there is for BSP.

The BSP model has been realized as a library of functions for process creation and destruction, remote memory access and message passing, and global, barrier-style, synchronization [5, 7, 4]. The abstraction offered by the BSP model is such that any library offering such facilities can be used for programming according to the BSP programming paradigm. The Oxford BSP Library [7] that supports Direct Remote Memory Access (DRMA) for parallel programs, the Green BSP library [5] that supports message passing and the Oxford BSP Toolset [4] that supports both DRMA and message passing are some of the libraries that specifically allow programming according to the BSP paradigm. Just as the von-Neumann model encompasses various programming language paradigms eg. functional, logic programming, the BSP does not dictate a particular mode of programming as well. All three libraries present a particular set of choices to the user. Some of the elements of this set are:

- *Data Parallel Program Structure.* It allows large-scale parallelism by splitting data and concurrently working on the individual pieces.
- *SPMD programs.* A Single Program Multiple Data programming style is used as perhaps implied by the structure of supersteps.
- *Direct mode of global memory management.* The programmer has direct access to memory allocation and determines how data are partitioned.
- *Static processor allocation.* The number of participating processors is determined in the beginning of the execution and cannot vary during the computation dynamically.

Various experimental studies of BSP algorithms have established that parallel algorithm design following the BSP paradigm can lead to a programming style which is easy to grasp and allows for the writing of programs that are scalable and transportable among diverse hardware platforms, without significant loss of performance. The same source code developed in one platform needs only to be recompiled on other ones to run efficiently without any modifications. The performance of BSP algorithms in diverse hardware platforms can be reliably predicted using the BSP parameters; one only needs an estimate of L and g for given p and the problem size n to predict algorithm performance for a particular machine. This way, one can reliably get a lower bound on the efficiency of a particular BSP algorithm implementation through the π and μ values.

Whereas performance of a particular BSP algorithm can be reliably predicted, one *should not* expect the BSP cost model to accurately predict the running time and behavior of a particular implementation. Accurate prediction is difficult even for sequential algorithms due to the existence of varying caching hierarchies; adding parallelism and the side-effects of communication introduces two more difficulties.

There have been various attempts to more accurately predict parallel performance by extending the BSP model. The E-BSP model is one such approach and is more explicit and specific about the communication network of a particular hardware platform, and the patterns of communication involved in routing. The attractiveness of the BSP cost model is its simplicity and generality; introducing more parameters to describe the performance of a communication network under various patterns of communication increases the complexity of describing the performance of all but the simplest algorithms with perhaps only small gains in prediction accuracy. Such an approach may also be problematic as it is the main reason parallel computing failed in the past: the attempt to realize for a particular algorithm those patterns of communication that are optimal for a given platform (and communication network), whereas they may lead in significant degradation in performance, if utilized in other platforms (portability vs. efficiency). Another variant of the BSP model introduces one more parameter, B, related to message size, and associates g with that message size, enforcing this way coarse-grained communication of messages of size equal to B. The original BSP model does not elaborate in detail between fine-grained and coarse-grained communication. If small h-relations are communicated (where "small" is to mean less than some parameter h_0, usually assumed to be equal to L/g) a cost L is assigned to such a communication; no mention of message size is inferred. Presumably, for BSP to be an abstract and general-purpose model, details of how communication is performed efficiently are left to the BSP library implementor. In practice, the generality of the BSP model works well if one interprets the cost model so as to absorb in L not only "small" h-relations but also "small" messages; therefore, the value of B is reflected in the choice of L and g (as is, h_0 as well) without the need of introducing an extra parameter.

References

[1] H. Dybdahl and I. Uthus. BSPlab programming environment. http://www.idi.ntnu.no/bsplab/, Norwegian University of Science and Technology, Norway, february 1998.

[2] A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel algorithms. *Journal of Parallel and Distributed Computing,* 22:251-267, 1994.

[3] A. V. Gerbessiotis and C. J. Siniolakis. Deterministic sorting and randomized median finding on the BSP model. In *Proceedings of the 8-th ACM Symposium on Parallel Algorithms and Architectures*, Padova, Italy, June 1996.

[4] M.W. Goudreau, J.M.D. Hill, K. Lang, W.F. McColl, S.D. Rao, D.C. Stefanescu, T. Suel, and T. Tsantilas. A proposal for a BSP Worldwide standard. BSP Worldwide, `http://www.bsp-worldwide.org/`, April 1996.

[5] M. Goudreau, K. Lang, S. Rao and T. Tsantilas. The Green BSP Library. Technical Report CR-TR-95-11, University of Central Florida, 1995.

[6] J. M. D. Hill. In `http://www.bsp-worldwide.org/implmnts/oxtool/`, September 1997.

[7] R. Miller. A library for Bulk-Synchronous Parallel programming. In *Proceedings of the British Computer Society Parallel Processing Specialist Group Workshop on General Purpose Parallel Computing*, December 1993.

[8] D. Talia. Parallel computation still not ready for the mainstream. *Communications of the ACM,* 44(7):98-99, July 1997.

[9] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM,* 33(8):103-111, August 1990.