TITLE

Bandwidth-Latency Models (BSP, LogP)

BYLINE

Thilo Kielmann
Department of Computer Science
Vrije Universiteit
Amsterdam
The Netherlands
kielmann@cs.vu.nl

Sergei Gorlatch
Department of Computer Science
Westfälische Wilhelms-Universität
Münster
Germany
gorlatch@uni-muenster.de

SYNONYMS

Parallel Communication Models
Message-passing Performance Models

DEFINITION

Bandwidth-latency models are a group of performance models for parallel programs that focus on modeling the communication between the processes in terms of network bandwidth and latency, allowing quite precise performance estimations. While originally developed for distributed-memory architectures, these models also apply to machines with non-uniform memory access (NUMA), like the modern multi-core architectures.

DISCUSSION

Introduction

The foremost goal of parallel programming is to speed up algorithms that would be too slow when executed sequentially. Achieving this so-called *speedup* requires a deep understanding of the performance of the inter-process communication and synchronization, together with the algorithm's computation. Both computation and communication/synchronization performance strongly depend on properties of the machine architecture in use. The strength of the bandwidth-latency models is that they model exactly the communication and synchronization operations. When the parameters of a given machine are fed into the performance analysis of a parallel algorithm, bandwidth-latency models can lead to rather precise and expressive performance evaluations. The most important models are BSP and LogP, as discussed below.

The BSP Model

The *bulk synchronous parallel* (BSP) model was proposed by Leslie Valiant[11] of Harvard University to overcome the shortcomings of the traditional PRAM model, while keeping its simplicity. None of the suggested PRAM models offers a satisfying forecast of the behavior of parallel machines for a wide range of applications. The BSP model (*bulk synchronous parallel*) was developed as a bridge between software and hardware developers: if the architecture of parallel machines is designed as prescribed by the BSP model, then software developers can rely on the BSP-like behaviour of the hardware. Furthermore it should not be necessary to customize perpetually the model of applications to new hardware details in order to benefit from a higher efficiency of emerging architectures.

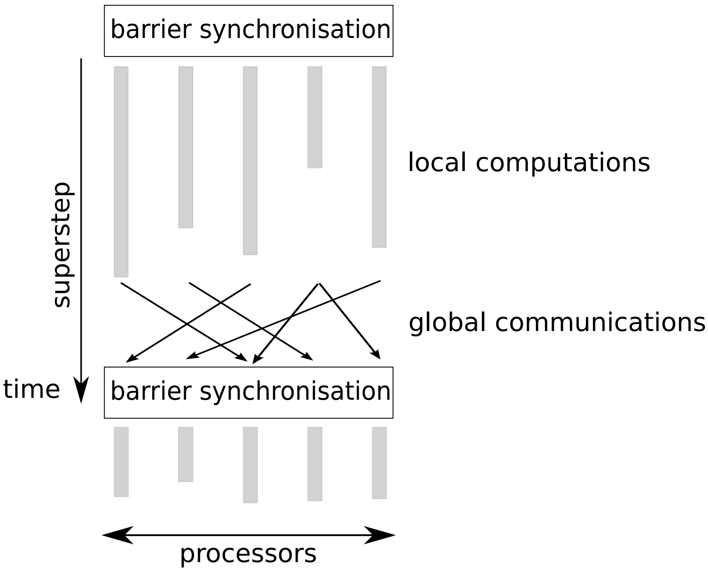

Figure 1: The BSP model executes calculations in supersteps. Each superstep comprises three phases: (1) simultaneous local computations performed by each process, (2) global communication operations for swapping data between processes, (3) barrier synchronisation for finalizing the communication operation and enabling access to received data by the receiving processes.

The BSP model is an abstraction of a machine with physically distributed memory that uses a presentation of communication as a global bundle instead of single point-to-point transfers. A BSP model machine consists of a number of processors equipped with memory, a connection network for point-to-point messages between processors, and a synchronisation mechanism, which allows a barrier synchronisation of all processors.

Calculations on a BSP machine are organized as a sequence of supersteps as shown in Figure 1:

- In each superstep, each processor executes local computations and may perform communication operations.

- A local computation can be executed in every time unit(step).

- The result of a communication operation does not become effective before the next superstep begins, i.e. the receiver cannot use received data until the current superstep is finished.

- At the end of every superstep, a barrier synchronisation using the synchronisation mechanism of the machine takes place.

The BSP model machine can be viewed as a MIMD system, because the processes can execute different instructions simultaneously. It is loosely synchronous at the superstep level, compared to the instruction-level tight synchrony in the PRAM model: within a superstep, different processes execute asynchronously at their own pace. There is a single address space, and a processor can access not only its local memory but also any remote memory in another processor, the latter imposing communication.

Within a superstep, each computation operation uses only data in the processor's local memory. These data are put into the local memory either at the program start-up time or by the communication operations of previous supersteps. Therefore, the computation operations of a process are independent of other processes, while it is not allowed for multiple processes to read or write the same memory location in the same step. Because of the barrier synchronization, all memory and communication operations in a superstep must completely finish before any operation of the next superstep begins. These restrictions imply that a BSP computer has a sequential consistency memory model.

A program execution on a BSP machine is characterised using the following four parameters[6]:

- p: the number of processors;
- s: the computation speed of processors expressed as the number of computation steps that can be executed by a processor per second. In each computation step, one arithmetic operation on local data can be executed;
- l: the number of steps needed for the barrier synchronisation;
- g: the average number of steps needed for transporting a word between two processors of the machine.

In a real parallel machine, there are many different patterns of communication between processors. For simplicity, the BSP model abstracts the communication operations using the h relation concept: an h relation is an abstraction of any communication operation, where each node sends at most h words to various nodes and each node receives at most h words. On a BSP computer, the time to realize any h relation is not longer than gh.

The BSP model is more realistic than the PRAM model, because it accounts for several overheads ignored by PRAM:

- To account for load imbalance, the computation time w is defined as the maximum number of steps spent on computation operations by any processor.
- The synchronization overhead is l, which has a lower bound of the communication network latency (i.e., the time for a word to propagate through the physical network) and is always greater than zero.
- The communication overhead is gh steps, i.e. gh is the time to execute the most time-consuming h relation. The value of g is platform-dependent: it is small on a computer with more efficient communication support.

The BSP model allows for overlapping of the computation, the communication, and the synchronization operations within a superstep. If all three types of operations are fully overlapped, then the time for a superstep becomes $max(w, gh, l)$. However, usually the more conservative $w + gh + l$ is used.

For a real parallel machine, the value of g depends on the bandwith of the communication network, the communication protocols, and the communication library. The value of l depends on the diameter of the network, as well as on the communication library. Both parameters are usually estimated using benchmark programs. Since the value s is used for normalization of the values l and g, only p, l and g are independent parameters. The execution time of a BSP-program is a sum of the execution time of all supersteps. The execution time of each superstep comprises three terms: (1) maximum local computation time of all processes, w, (2) costs of global communication realizing a h-relation, and (3) costs for the barrier synchronisation finalizing the superstep:

$$T_{superstep} = w + h \cdot g + l \tag{1}$$

The BSP model was implemented in a so-called BSPlib library [5, 6] that provides operations for initialisation of supersteps, execution of communication operations, and barrier synchronisations.

The LogP Model

In [4], Culler et al. criticise BSP's assumption that the length of supersteps must allow realizing arbitrary h-relations which means that the granularity has a lower bound. Also the messages that have been sent during a superstep are not available for a recipient before the following superstep begins, even if a message is sent and received within the same superstep. Furthermore, the BSP model assumes hardware support for the synchronisation mechanism, although most existing parallel machines do not provide such a support. Because of these problems, the LogP model [4] has been devised, that is arguably more closely related to the modern hardware used in parallel machines.

In analogy to BSP, the LogP model assumes that a parallel machine consists of a number of processors equipped with memory. The processors can communicate using point-to-point messages through a communication network. The behaviour of the communication is described using four parameters L, o, g, and P, which give the model the name LogP:

- L (latency) is an upper bound for the network's latency, i.e. the maximum delay between sending and receiving a message;
- o (overhead) describes the time a processor is busy with sending or receiving a message; during this time, no other calculations can be done by that processor;
- g (gap) denotes the minimal timespan between sending or receiving two messages back-to-back;
- P is the number of processors of the parallel machine.

Figure 2 shows a visualization of the parameters [3]. Except P, all parameters are measured in time units or multiple machine cycle units. The model also

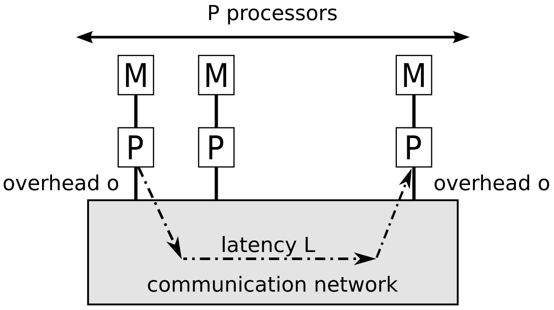

Figure 2: Parameter visualization of the LogP model

assumes a network with finite capacity. From the definitions of L and g follows that, for a given pair of source and destination nodes, the number of messages that can be on their way through the network simultaneously is limited by L/g. A processor trying to send a message which would exceed this limitation will be blocked until the network can accept the next message. This property models the network bandwidth where the parameter g reflects the bottleneck bandwidth, independent of the bottleneck's location, be it the network link, or be it the processing time spent sending or receiving. g thus denotes an upper bound for o.

For this capacity constraint, LogP both gets praised and criticised. The advantage of this constraint is a very realistic modeling of communication performance, as the bandwidth capacity of a communication path can get limited by any entity between the sender's memory and the receiver's memory. Due to this focus on point-to-point communication, LogP (variants) have been successfully used for modeling various computer communication problems, like the performance of the Network File System (NFS) [9] or collective communication operations from the Message Passing Interface (MPI) [8]. The disadvantage of the capacity constraint is that LogP exposes the sensitivity of a parallel computation to the communication performance of individual processors. This way, analytic performance modeling is much harder with LogP, compared to models with higher abstraction level like BSP [2].

While LogP assumes that the processors are working asynchronously, it requires that all messages must not exceed a preset size, i.e. larger messages must be fragmented. The latency of a single message is not predictable, but it is limited by L. This means, in particular, that messages can overtake each other such that the recipient may potentially receive the messages out of order. The values of the parameters L, o and g depend on hardware characteristics, the used communication software, and the underlying communication protocols.

The time to communicate a message from one node to another (i.e., the start-up time t_0) consists of three terms: $t_0 = o + L + o$. The first o is called the *send overhead*, which is the time at the sending node to execute a message send operation to inject a message into the network. The second o is called the *receive overhead*, which is the time at the receiving node to execute a message receive operation. For simplicity, the two overheads are assumed equal and called the *overhead* o, i.e. o is the length of a time period that a node is engaged in sending or receiving a message. During this time, the node cannot perform other operations (e.g., overlapping computations).

In the LogP-model the runtime of an algorithm is determined as the maximum runtime across all processors. A consequence of the LogP model is that the access to a data element in memory of another processor costs $T_m = 2L + 4o$ time units (a message rountrip), of which one half is used for reading, and the other half for writing. A sequence of n pipelined messages can be delivered in $T_n = L + 2o + (n - 1)g$ time units, as shown in Figure 3.

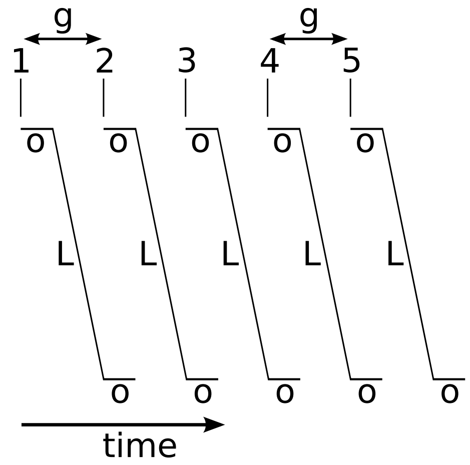

Figure 3: Modeling the transfer of a large message in n segments using the LogP model. The last message segment is sent at time $T_s = (n - 1) \cdot g$ and will be received at time $T_r = T_s + 2o + L$

A strong point of LogP is its simplicity. This simplicity, however, can, at times, lead to inaccurate performance modeling. The most obvious limitation is the restriction to small messages only. This was overcome by introducing a LogP variant, called LogGP [1]. It contains an additional parameter G (Gap per Byte) as the time needed to send a byte from a large message. $1/G$ is the bandwidth per processor. The time for sending a message consisting of n Bytes is then $T_n = o + (n - 1)G + L + o$.

A more radical extension is the *parameterized LogP* model [7], PLogP, for short. PLogP has been designed taking observations with communication software, like the Message Passing Interface (MPI), into account. These observations are: (1.) Overhead and gap strongly depend on the message size; some communication libraries even switch between different transfer implementations, for short, medium, and large messages. (2.) Send and receive overhead can strongly differ, as the handling of asynchronously incoming messages needs a fundamentally different implementation than a synchronously invoked send operation. In the PLogP model, the original parameters o and g have been replaced by the *send overhead* $o_s(m)$, the *receive overhead* $o_r(m)$, and the *gap* $g(m)$, where m is the message size. L and P remain the same as in LogP. Figure 4 illustrates this.

PLogP allows precise performance modeling when parameters for the relevant message sizes of an application are used. In [8], this has been demonstrated for MPI's collective communication operations, even for hierarchical communi-

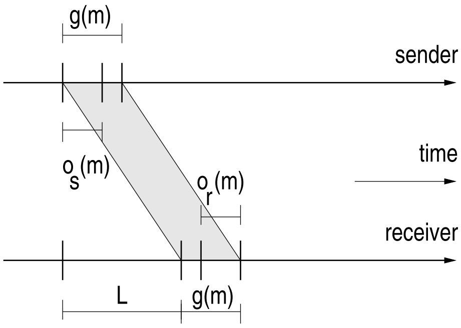

Figure 4: Visualization of a message transport with m bytes using the PLogP model. The sender is busy for $o_s(m)$ time. The message has been received at $T = L + g(m)$, out of which the receiver had been busy for $o_r(m)$ time.

cation networks with different sets of performance parameters. Pješivac-Grbović et al. [10] have shown that PLogP provides flexible and accurate performance modeling.

Concluding Remarks

Historically, the major focus of parallel algorithm development had been the PRAM model, which ignores data access and communication cost and considers only load balance and extra work. The PRAM model is very useful in understanding the inherent concurrency in an application, which is the first conceptual step in developing a parallel program; however, it does not take important realities of particular systems into account, such as the fact that data access and communication costs are often the dominant components of execution time.

The bandwidth-latency models described in this chapter articulate the performance issues against which software must be designed. Based on a clearer understanding of the importance of communication costs on modern machines, models like BSP and LogP help analyze communication cost and hence improve the structure of communication. These models expose the important costs associated with a communication event, such as latency, bandwidth, or overhead, allowing algorithm designers to factor them into the comparative analysis of parallel algorithms. Even more, the emphasis on modeling communication cost has shifted to the cost at the nodes that are the endpoints of the communication message, such that the number of messages and contention at the endpoints have become more important than mapping to network technologies. In fact, both the BSP and LogP models ignore network topology, modeling network delay as a constant value.

An in-depth comparison of BSP and LogP has been performed in [2], showing that both models are roughly equivalent in terms of expressiveness, slightly favoring BSP for its higher-level abstraction. But it was exactly this model that was found to be too restrictive by the designers of LogP [4]. Both models have their advantages and disadvantages. LogP is better suited for modeling applications that actually use point-to-point communication, while BSP is better and simpler for data-parallel applications that fit the superstep model. The BSP model also provides an elegant framework that can be used to reason about communication and parallel performance. The major contribution of both models is the explicit acknowledgement of communication costs that are dependent on properties of the underlying machine architecture.

The BSP and LogP models are important steps toward a realistic architectural model for designing and analyzing parallel algorithms. By experimenting with the values of the key parameters in the models, it is possible to determine how an algorithm would perform across a range of architectures and how it might be best structured for different architectures or for portable performance.

RELATED ENTRIES

To be written when the list of topics covered in the book will become available.

BIBLIOGRAPHIC NOTES AND FURTHER READING

The presented models and their extensions were originally introduced in papers [1, 4, 11] and described in textbooks [12, 13, 14] which were partially used for writing this entry. Further papers in the list of references deal with particular applications of the models and their classification and standardisation.

References

[1] A. Alexandrov, M. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model – One Step Closer towards a Realistic Model for Parallel Computation. In *7th ACM Symposium on Parallel Algorithms and Architectures(SPAA'95)*, pages 95–105, Santa Barbara, California, July 1995.

[2] G. Bilardi, K. T. Herley, A. Pietracaprina, G. Pucci, and P. Spirakis. BSP versus LogP. *Algorithmica*, Vol. 24, pp. 405-422, 1999.

[3] D. E. Culler, A. C. Dusseau, R. P. Martin, and K. E. Schauser. Fast Parallel Sorting under LogP: from Theory to Practice. In *Portability and Performance for Parallel Processing*, pages 71–98, Wiley, 1994.

[4] D. E. Culler, R. Karp, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken LogP: Towards a Realistic Model of Parallel Computation. *4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'93)*, pages 1–12, 1993.

[5] M. W. Goudreau, J. M. Hill, K. Lang, W. F. McColl, S. D. Rao, D. C. Stefanescu, T. Suel, and T. Tsantilas. A proposal for a BSP Worldwide standard. Technical Report, BSP Worldwide, www.bsp-wordwide.org, 1996.

[6] M. Hill, W. McColl, and D. Skillicorn. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[7] T. Kielmann, H. E. Bal, and K. Verstoep. Fast Measurement of LogP Parameters for Message Passing Platforms. *4th Workshop on Runtime Systems for Parallel Programming (RTSPP)*, held in conjunction with IPDPS 2000, May 2000.

[8] T. Kielmann, H. E. Bal, S. Gorlatch, K. Verstoep, and R. F. H. Hofman. Network Performance-aware Collective Communication for Clustered Wide Area Systems. *Parallel Computing*, Vol. 27, No. 11, pp. 1431-1456, 2001.

[9] R. P. Martin and D. E. Culler. NFS Sensitivity to High Performance Networks. *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 71-82, 1999.

[10] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance Analysis of MPI Collective Operations. *4th Int. Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS'05)*, April 2004.

[11] L. G. Valiant. A Bridging Model for parallel Computation. *Communications of the ACM*,33(8):103–111, 1990.

[12] T. Rauber, and G. Rünger. *Parallel Programming: for Multicore and Cluster Systems*. Springer, 2010.

[13] K. Hwang, and Z. Xu. *Scalable Parallel Computing*, WCB/McGraw-Hill, 1998.

[14] D. E. Culler, and J. P. Singh, and A. Gupta. *Parallel Computer Architecture – A Hardware/Software Approach*, Morgan Kaufmann Publishers, 1999