Bulk synchronous parallel

The Bulk Synchronous Parallel (BSP) abstract computer is a bridging model for designing parallel algorithms. It serves a purpose similar to the Parallel Random Access Machine (PRAM) model. BSP differs from PRAM by not taking communication and synchronization for granted. An important part of analysing a BSP algorithm rests on quantifying the synchronization and communication needed.

History

The BSP model was developed by Leslie Valiant of Harvard University during the 1980s. The definitive article[1] was published in 1990.

Between 1990 and 1992, Leslie Valiant and Bill McColl of Oxford University worked on ideas for a distributed memory BSP programming model, in Princeton and at Harvard. Between 1992 and 1997, McColl led a large research team at Oxford that developed various BSP programming libraries, languages and tools, and also numerous massively parallel BSP algorithms. With interest and momentum growing, McColl then led a group from Oxford, Harvard, Florida, Princeton, Bell Labs, Columbia and Utrecht that developed and published the BSPlib Standard for BSP programming in 1996.

Valiant developed an extension to the BSP model in the 2000s, leading to the publication of the Multi-BSP model[2] in 2011.

The model

A BSP computer consists of

1. components capable of processing and/or local memory transactions (i.e., processors),
2. a network that routes messages between pairs of such components, and
3. a hardware facility that allows for the synchronisation of all or a subset of components.

This is commonly interpreted as a set of processors which may follow different threads of computation, with each processor equipped with fast local memory and interconnected by a communication network. A BSP algorithm relies heavily on the third feature; a computation proceeds in a series of global supersteps, which consists of three components:

- **Concurrent computation**: every participating processor may perform local computations, i.e., each process can only make use of values stored in the local fast memory of the processor. The computations occur asynchronously of all the others but may overlap with communication.
- **Communication**: The processes exchange data between themselves to facilitate remote data storage capabilities.
- **Barrier synchronisation**: When a process reaches this point (the barrier), it waits until all other processes have reached the same barrier.

The computation and communication actions do not have to be ordered in time. Communication typically takes the form of the one-sided put and get Direct Remote Memory Access (DRMA) calls, rather than paired two-sided send and receive message passing calls. The barrier synchronization concludes the superstep: it ensures that all one-sided communications are properly concluded. Systems based on two-sided communication include this synchronisation cost implicitly for every message sent. The method for barrier synchronisation relies on the hardware facility of the BSP computer. In Valiant's original paper, this facility periodically checks if the end of the current superstep is reached globally. The period of this check is denoted by $L$.

The figure below shows this in a diagrammatic form. The processes are not regarded as having a particular linear order (from left to right or otherwise), and may be mapped to processors in any way.
The BSP model is also well-suited to enable automatic memory management for distributed-memory computing through overdecomposition of the problem and oversubscription of the processors. The computation is divided into more logical processes than there are physical processors, and processes are randomly assigned to processors. This strategy can be shown statistically to lead to almost perfect load balancing, both of work and communication.

**Communication**

In many parallel programming systems, communications are considered at the level of individual actions: sending and receiving a message, memory to memory transfer, etc. This is difficult to work with, since there are many simultaneous communication actions in a parallel program, and their interactions are typically complex. In particular, it is difficult to say much about the time any single communication action will take to complete.

The BSP model considers communication actions *en masse*. This has the effect that an upper bound on the time taken to communicate a set of data can be given. BSP considers all communication actions of a superstep as one unit, and assumes all individual messages sent as part of this unit have a fixed size.

The maximum number of incoming or outgoing messages for a superstep is denoted by $h$. The ability of a communication network to deliver data is captured by a parameter $g$, defined such that it takes time $hg$ for a processor to deliver $h$ messages of size 1.

A message of length $m$ obviously takes longer to send than a message of size 1. However, the BSP model does not make a distinction between a message length of $m$ or $m$ messages of length 1. In either case the cost is said to be $mg$.

The parameter $g$ is dependent on the following factors:

- The protocols used to interact within the communication network.
- Buffer management by both the processors and the communication network.
- The routing strategy used in the network.
- The BSP runtime system.

In practice, $g$ is determined empirically for each parallel computer. Note that $g$ is not the normalised single-word delivery time, but the single-word delivery time under continuous traffic conditions.
Barriers

The one-sided communication of the BSP model requires barrier synchronization. Barriers are potentially costly, but avoid the possibility of deadlock or livelock, since barriers cannot create circular data dependencies. Tools to detect them and deal with them are unnecessary. Barriers also permit novel forms of fault tolerance.

The cost of barrier synchronization is influenced by a couple of issues:

1. The cost imposed by the variation in the completion time of the participating concurrent computations. Take the example where all but one of the processes have completed their work for this superstep, and are waiting for the last process, which still has a lot of work to complete. The best that an implementation can do is ensure that each process works on roughly the same problem size.

2. The cost of reaching a globally consistent state in all of the processors. This depends on the communication network, but also on whether there is special-purpose hardware available for synchronizing, and on the way in which interrupts are handled by processors.

The cost of a barrier synchronization is denoted by $l$. Note that $l < L$ if the synchronization mechanism of the BSP computer is as suggested by Valiant. In practice, a value of $l$ is determined empirically.

On large computers barriers are expensive, and this is increasingly so on large scales. There is a large body of literature on removing synchronization points from existing algorithms, both in the context of BSP computing and beyond. For example, many algorithms allow for the local detection of the global end of a superstep simply by comparing local information to the number of messages already received. This drives the cost of a global synchronization, compared to the minimally required latency of communication, to zero. Yet also this minimal latency is expected to increase further for future supercomputer architectures and network interconnects; the BSP model, along with other models for parallel computation, require adaptation to cope with this trend. Multi-BSP is one BSP-based solution.

The Cost of a BSP algorithm

The cost of a superstep is determined as the sum of three terms; the cost of the longest running local computation, the cost of global communication between the processors, and the cost of the barrier synchronisation at the end of the superstep. The cost of one superstep for $p$ processors:

$$\max_{i=1}^{p}(w_i) + \max_{i=1}^{p}(h_i g) + l$$

where $w_i$ is the cost for the local computation in process $i$, and $h_i$ is the number of messages sent or received by process $i$. Note that homogeneous processors are assumed here. It is more common for the expression to be written as $w + h g + l$ where $w$ and $h$ are maxima. The cost of the algorithm then, is the sum of the costs of each superstep:

$$W + H g + S l = \sum_{s=1}^{S} w_s + g \sum_{s=1}^{S} h_s + S l$$

where $S$ is the number of supersteps.

The cost $W$, $H$, and $S$ are usually modelled as functions, that vary with problem size. These three characteristics of a BSP algorithm are usually described in terms of asymptotic notation, e.g. $H \in O(n/p)$.

Extensions and uses

Interest in BSP has soared in recent years, with Google adopting it as a major technology for graph analytics at massive scale via technologies like Pregel and MapReduce. Also, with the next generation of Hadoop decoupling the MapReduce model from the rest of the Hadoop infrastructure, there are now active open source projects to add explicit BSP programming, as well as other high performance parallel programming models, on top of Hadoop. Examples are Apache Hama and Apache Giraph.
BSP has been extended by many authors to address concerns about BSP's unsuitability for modelling specific architectures or computational paradigms. One example of this is the decomposable BSP model. The model has also been used in the creation of a number of new programming languages and interfaces, such as Bulk Synchronous Parallel ML (BSML), BSPlib,[5] Apache Hama, and Pregel.[6]

Notable implementations of the BSPlib standard are the Paderborn University BSP library[7] and the Oxford BSP Toolset by Jonathan Hill.[8] Modern implementations include BSPonMPI[9] (which simulates BSP on top of the Message Passing Interface), and MulticoreBSP[10][11] (a novel implementation targeting modern shared-memory architectures). MulticoreBSP for C is especially notable for its capability of starting nested BSP runs, thus allowing for explicit Multi-BSP programming.

References


External links

• Collection of papers on BSP by Bill McColl (http://www.linkedin.com/in/billmccoll) and others. BSP Papers (http://paloaltodata.com/index.php?option=com_content&view=article&id=22)
• BSP Worldwide (http://www.bsp-worldwide.org/)
• BSP related papers (http://www.bsp-worldwide.org/implmnts/oxtool/papers.html)
• (French) Bulk Synchronous Parallel ML ((English) official website (http://frederic.loulergue.eu/research/bsml/index.html))
• Apache Hama (http://hama.apache.org/)
• Apache Giraph (http://giraph.apache.org/)
• Paderborn University BSP library (http://www2.cs.uni-paderborn.de/~pub/)
• BSPonMPI (http://bsponmpi.sourceforge.net)
• MulticoreBSP (http://www.multicorebsp.com)
Parallel random-access machine

In computer science, a parallel random-access machine (PRAM) is a shared-memory abstract machine. As its name indicates, the PRAM was intended as the parallel-computing analogy to the random-access machine (RAM). In the same way that the RAM is used by sequential-algorithm designers to model algorithmic performance (such as time complexity), the PRAM is used by parallel-algorithm designers to model parallel algorithmic performance (such as time complexity, where the number of processors assumed is typically also stated). Similar to the way in which the RAM model neglects practical issues, such as access time to cache memory versus main memory, the PRAM model neglects such issues as synchronization and communication, but provides any (problem-size-dependent) number of processors. Algorithm cost, for instance, is estimated using two parameters \(O(\text{time})\) and \(O(\text{time} \times \text{processor\_number})\).

Read/write conflicts

Read/write conflicts in accessing the same shared memory location simultaneously are resolved by one of the following strategies:

1. Exclusive read exclusive write (EREW)—every memory cell can be read or written to by only one processor at a time
2. Concurrent read exclusive write (CREW)—multiple processors can read a memory cell but only one can write at a time
3. Exclusive read concurrent write (ERCW)—never considered
4. Concurrent read concurrent write (CRCW)—multiple processors can read and write. A CRCW PRAM is sometimes called a concurrent random-access machine.[1]

Here, E and C stand for 'exclusive' and 'concurrent' respectively. The read causes no discrepancies while the concurrent write is further defined as:

- **Common**—all processors write the same value; otherwise is illegal
- **Arbitrary**—only one arbitrary attempt is successful, others retire
- **Priority**—processor rank indicates who gets to write

Another kind of array reduction operation like SUM, Logical AND or MAX.

Several simplifying assumptions are made while considering the development of algorithms for PRAM. They are:

1. There is no limit on the number of processors in the machine.
2. Any memory location is uniformly accessible from any processor.
3. There is no limit on the amount of shared memory in the system.
4. Resource contention is absent.
5. The programs written on these machines are, in general, of type SIMD.

These kinds of algorithms are useful for understanding the exploitation of concurrency, dividing the original problem into similar sub-problems and solving them in parallel.
Implementation

PRAM algorithms cannot be parallelized with the combination of CPU and dynamic random-access memory (DRAM) because DRAM does not allow concurrent access; but they can be implemented in hardware or read/write to the internal static random-access memory (SRAM) blocks of a field-programmable gate array (FPGA), it can be done using a CRCW algorithm.

However, the test for practical relevance of PRAM (or RAM) algorithms depends on whether their cost model provides an effective abstraction of some computer; the structure of that computer can be quite different than the abstract model. The knowledge of the layers of software and hardware that need to be inserted is beyond the scope of this article. But, articles such as Vishkin (2011) demonstrate how a PRAM-like abstraction can be supported by the explicit multi-threading (XMT) paradigm and articles such as Caragea & Vishkin (2011) demonstrate that a PRAM algorithm for the maximum flow problem can provide strong speedups relative to the fastest serial program for the same problem.

Example code

This is an example of SystemVerilog code which finds the maximum value in the array in only 2 clock cycles. It compares all the combinations of the elements in the array at the first clock, and merges the result at the second clock. It uses CRCW memory; \( m[i] <= 1 \) and \( \text{maxNo} <= \text{data[i]} \) are written concurrently. The concurrency causes no conflicts because the algorithm guarantees that the same value is written to the same memory. This code can be run on FPGA hardware.

```verilog
module FindMax #(parameter int len = 8)
  (input bit clock, resetN, input bit[7:0] data[len], output bit[7:0] maxNo);

typedef enum bit[1:0] {COMPARE, MERGE, DONE} State;

State state;
bit m[len];
int i, j;

always_ff @(posedge clock, negedge resetN) begin
  if (!resetN) begin
    for (i = 0; i < len; i++) m[i] <= 0;
    state <= COMPARE;
  end else begin
    case (state)
      COMPARE: begin
        for (i = 0; i < len; i++) begin
          for (j = 0; j < len; j++) begin
            if (data[i] < data[j]) m[i] <= 1;
          end
        end
        state <= MERGE;
      end
      MERGE: begin
        for (i = 0; i < len; i++) begin
          if (m[i] == 0) maxNo <= data[i];
        end
      end
    endcase
  end
end
```
Parallel random-access machine

```plaintext
state <= DONE;
end
case
end
endmodule
```

**External links**

- Saarland University's prototype PRAM [2]
- University Of Maryland's PRAM-On-Chip prototype [3]. This prototype seeks to put many parallel processors and the fabric for inter-connecting them on a single chip

**References**


Vishkin, Uzi (2009), *Thinking in Parallel: Some Basic Data-Parallel Algorithms and Techniques*, 104 pages (http://www.umiacs.umd.edu/users/vishkin/PUBLICATIONS/classnotes.pdf), Class notes of courses on parallel algorithms taught since 1992 at the University of Maryland, College Park, Tel Aviv University and the Technion.


Image Sources, Licenses and Contributors
