Parallel Programming Languages and Systems
Exercise 2

s0840449

1 Critique of the BSP Model

1.1 Key Concepts

The key concept behind the Bulk Synchronous Parallel model (BSP) is the partitioning of process execution into discrete computational units, named *supersteps*. Each superstep is divided into three phases:

1. **Local Computation**: the process performs computation that does not require it to communicate with any other process. Usually the process would be expected to perform as much of such computation as possible, although that is not a requirement.

2. **Global Communication**: processes that wish to communicate send *one-way* messages to other processes.

3. **Global Synchronization**: processes wait at a global barrier until every process has arrived. Once all processes are at the barrier the current superstep is over and the next begins.

The rational behind the BSP model is that tracking individual communication between processes (either by passing messages or by sharing memory) is the cause of most parallel programming difficulties. In BSP all communication between processes takes place 'at once', during the Global Communication step. This separates out the communication from the computation, avoiding common problems such as message deadlock (when two processes both wait on each other to send a message) and concurrent access to data (e.g. a writer process updating a piece of data before another reader process has seen it.) The Global Synchronization step ensures that no process races ahead and pollutes the current communication.

It is important to note that the breakdown of a superstep is conceptual only. Similar to the sequential consistency model seen in PPLS, BSP requires only that the implementation of supersteps is semantically equivalent to the above. For example, when and how communication is done is irrelevant, as long as it does not influence the Local Computation in the same superstep (which semantically comes before it) and as long as it is completed before the next superstep.

By discretising parallel program execution, the BSP model is also able to provide a 'one size fits all' approach to measuring program performance. The cost of a BSP computation can be neatly written as

$$cost = \sum_{s \in supersteps} cost(s),$$

where the performance of each superstep is given by

$$cost(superstep) = max_i(w_i) + max_i(h_i g) + l$$

The three values in the superstep cost formula correspond to the cost of each of the sub-steps. The cost of the local computation performed by a process i is captured by w_i; a maximum is then taken across all the processes to find the most expensive local computation. The next value, $max_i(h_i g)$, captures the cost of the global communication, in terms of message bottleneck (the maximum of the incoming and outgoing messages for each process i, h_i) and the overall ability of the communication network (g). Note that h_i refers only to the number of messages sent; to simplify analysis, BSP assumes a fixed message size (based on the largest message sent). This assumption has been challenged in the literature: [1] argues that in practice the message size is impactful enough that it should be explicitly considered. Finally, the cost of synchronizing the processes during the Global Synchronization sub-step is captured by l.

The use of a single formula for cost allows the comparison of both different BSP algorithms for a problem (which will have different w_i, h_i, and a different number of supersteps), and different architectures for a single known BSP algorithm (which will have different values of g and l). The formula also exposes some of the conceptual problems with BSP: the importance of balancing the local computation performed by each process (due to the $max(w_i)$ coefficient), and the requirement that all processes perform the same number of supersteps. These are discussed further below.

1.2 Pipeline Computations in BSP

The BSP model is well suited for programming pipelined parallel computations. One of the main drawbacks of the BSP model is that the overall computation waits on the slowest process at each superstep. However in a pipelined computation both the latency and (more importantly) the throughput are restricted by the slowest stage in the pipeline:

$$latency = N * max(time), \text{ and}$$

$$throughput = latency + (max(time) * (N - 1)),$$

where the maximums are performed over the pipeline stages, and N is the number of stages. This means that the wait enforced by the BSP model is not a problem in a well designed pipeline algorithm, as each stage will be well balanced anyway. In addition to balanced computations, pipelined algorithms usually have well balanced communication costs, as each stage (process) receives only from its predecessor and sends only to its successor. This again fits nicely for the BSP model, the performance of which depends also on the worst-case communication each superstep. If the communications are not balanced (for example, if one stage expands its input into a set of larger pieces to send to its successor), again the imbalance would affect the throughput in a standard pipeline algorithm, so any problem would not be caused by the use of the BSP model.

There are some potential drawbacks to using the BSP model for pipelined parallel computation. Firstly, all BSP processes must execute some code every superstep. This means that pipeline stages without an input still have to be scheduled and do some work every superstep. Such wasted effort will be especially noticeable when filling or flushing the pipeline (at the beginning and end of computation), as most stages will be empty. Additionally, some pipeline computations (such as the Sieve of Eratosthenes) necessitate the creation of new processes on demand. It is not entirely clear whether the BSP model supports dynamic process creation (it is quite possibly considered an implementation issue).

A final nice aspect of the BSP model for pipelined parallel computations is the applicability of the BSP performance formula. One of the most important decisions to make when implementing a pipeline computation is balancing the computation of each stage with the cost of communication - having too many stages doing very little work is wasteful. The BSP formula could be used to try and balance the computation and communication in the pipeline, as it is able to break down the cost of computation with regards to those units.

1.3 Bag Of Tasks Computations in BSP

The BSP model is not well suited for programming bag of tasks (BoT) computations. The BoT paradigm is designed to be flexible, allowing tasks to be processed whenever they become available, and results to be collected similarly. This conflicts with the BSP model, which is designed around structured computation. The foremost problem is the use of supersteps in BSP, and the lock-step style computation they entail. In BoT, tasks can be given to workers at any time, can take varying lengths of time to process (i.e. some tasks may involve more work than others), and results can be sent back to the controlling process at any time. In a BSP implementation, however, all processes would have to wait for the slowest current worker to finish processing (or to reach some sort of defined intermediary step; this might alleviate the problem but would complicate the code.) Due to this, a single long-running task can hold the computation 'hostage', preventing other works from reporting their results and moving on with any new computations.

In addition to the computational problems, a BSP-based BoT may struggle with communication. BoT naturally has a communication bottleneck at the farmer (or whichever process is responsible for holding the bag), as all workers communicate with it. In a standard BoT, however, the communication is usually spread out temporally (as workers start and finish), alleviating the problem. With the BSP model, all communication takes place at once, and as discussed above workers will have to finish in batches (defined by the slowest worker). In this case, the farmer becomes a bottleneck. For P processes, in the worst case the farmer would be a $(P-1)$-relation (as it might receive $P-1$ messages in one superstep). Note that the bottleneck is likely only a conceptual problem: in practice many BSP libraries choose to spread communication throughout the superstep, and so temporal effects may still alleviate the bottleneck problem.

References

[1] Peter Krusche. Experimental evaluation of bsp programming libraries. *Parallel Processing Letters*, 18(01):7–21, 2008.