SPM Project: BSP

Lorenzo Bellomo, 531423

AA 2018/2019

1 Introduction

In this report the *Bulk Synchronous Pattern* (BSP) will be analysed both from a theoretical viewpoint and from a practical one.
The outline of the report will be the following:

- Section 2 (Parallel Architecture Design)
- Section 3 (Performance Modeling)
- Section 4 (Implementation Structure and Details)
- Section 5 (Experimental Evaluation)

2 Parallel Architecture Design

BSP is by nature a parallel bridging model. The main idea is that the process is divided in three phases (as shown in figure 1):

- *Local computation*: The processors act independently and compute their task on a different partition of the input data.
- *Communication*: In this phase processors are allowed to send data to the other ones. It is important to notice that phase 1 and 2 (super step and communication) may overlap in case of uneven input workload.
- *Synchronization*: In this phase all the processors synchronize and wait that the communication phase is over for each worker (*barrier*).

This whole process made of 3 phases composes a *super step*.

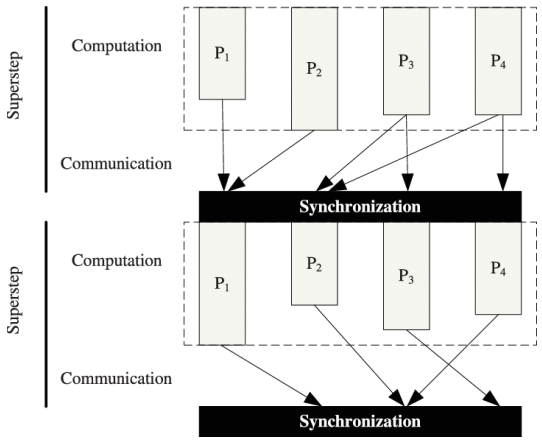

Figure 1: BSP structure

3 Performance Modeling

In order to theoretically model the performances of a BSP algorithm some values are going to be defined:

- nw is the number of workers (BSP processors).
- k is the number of super steps
- w_i is the time spent by the *i-th* worker in the compute phase.
- m_i is the number of maximum messages receiver by the *i-th* worker.
- c is the cost of sending or receiving one message.
- B is the cost of the synchronization step (*barrier*).
- I is the initialization process cost.
- F is the finalization process cost

The cost of a super step (SS) can be approximated (upper bounded) as follows:

$$SS_j = \max_{i=1}^{nw} w_i + \max_{i=1}^{nw} cm_i + B$$

Which eventually gives:

$$tot\_cost = I + \sum_{j=1}^{k} SS_j + F$$

3.1 Opportunities for parallelism

According to the model provided, the opportunities for parallelization reside in the *super step cost*, which ideally should decrease linearly by increasing the number of parallel activities. In particular, it is safe to assume that $w_i = \Theta(n/nw)\,\forall i$, while instead both the communication cost and the barrier cost are pure overhead, which grows with nw.
In this particular problem, the parallel design is already provided, but the goal of the implementation is to minimize the overheads, with particular focus on the *communication* one, which can be assumed to be the most costly.

For what regards the user provided business logic code, which regards both the computation phase and the communication one, the scalability of the provided *BSP* depends on the goodness of the business logic code.

4 Implementation Structure and Details

All the source code files are under directory *src* and are provided under the form of header only files.
The files provided are:

- `sorter.cpp`: it contains the main method and the *FastFlow* business logic code.
- `sorterLogic.hpp`: it contains the business logic code for the custom *BSP* execution with standard *POSIX* threads.
- `posixBSP.hpp`: it contains the implementation (with *POSIX* threads) of the BSP pattern.

- `logicBSP.hpp`: it contains the interface that hosts the business logic code abstraction. Every application should subclass the one provided in this file to provide the business logic code. More in section 4.3.
- `barrier.hpp`: it contains an implementation of a reusable (context aware) barrier, implemented with a mutex and a condition variable.
- `safeQueue.hpp`: it contains the queue implementation used. Its discussion will be delayed to section 4.1.
- `queueMatrix.hpp`: It contains a class to handle a matrix of queues. Its use will be to give to every worker a queue *for each* super step. The matrix dimension will be $matrix[\#ss][nw]$.
- `utimer.hpp`
- `makefile`: the only thing worth noting is that the code is optimized with `-O3`, and the *FastFlow* include path might have to be changed. The one provided works with the installation on the remote *XEON PHI* machine. Just issuing `make all` should work.
- *benchmark folder*: The analysis of the content of this folder is postponed to section 5.

4.1 Communications

The queue implementation is a modification of the one provided during classes. The main difference is that writes to the queue are synchronized, while reads are not.
This is made possible by the fact that reads are always performed after a barrier, which makes the synchronization implicit. Writes have to be executed in mutual exclusion because multiple producers may potentially write together. In order to reduce the syncronization overhead, it is suggested to use the method `push_multiple(iterator, iterator)`, in order to lock once the queue and then release it immediately.

4.2 Threads

The code of the threads is a private function (`worker`) in file `posixBSP.hpp`.
It should be noted that the program tries to stick the worker threads to one core in increasing order (thread 0 to core 0, thread 1 to core 1...) with `pthread_setaffinity_np`. If this process fails then the computation goes on without sticking the threads to cores.

4.3 Business Logic Code

In order to write code that can be fed to the provided BSP implementation, one has to subclass `logicBSP`. This is an interface that models a generic worker (see `sorterLogic.hpp` for an example).

One has to provide the code of the worker's *super steps* (every worker has its own local state) and it must provide a switcher function that maps every function to a super step number.
The code of a generic super step has to be a function with this signature:

```
void ss(logicBSP::ss_queue, size_t, std::vector<logicBSP::ss_queue>)
```

4.4 FastFlow Implementation

5 Experimental Evaluation