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 Modelling).
- Section 4 (Implementation Structure and Details).
- Section 5 (Experimental Evaluation).
- Section 6 (Conclusions).

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*.

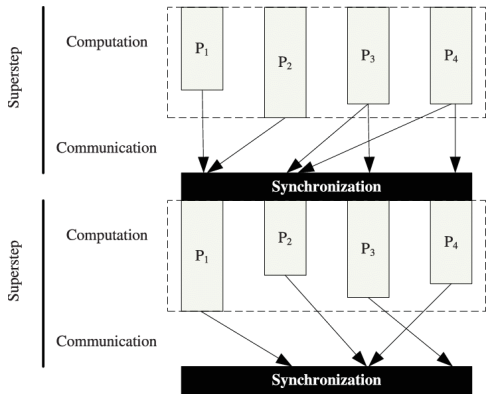

Figure 1: BSP structure

3 Performance Modelling

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 and F are relatively the initialization and finalization cost. Their value is application specific.

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.
- `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`
- `benchmark`: This folder will be analysed in section 5.1.

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 synchronization 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`.
As explained, one thread executes the code of one worker (which may comprise more super steps), so in total only nw threads are spawned by the runtime.

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 (except for the function name, which might vary):

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

Where `logicBSP::ss_queue` is just an alias for a pointer to a `safe_queue<T>`. The runtime will give the handle to the workers' *input queue* and the collection of *output_queues*, indexed exactly like the workers (where `output_queues[i]` is the queue belonging to worker `i`).
This way, every worker keeps its own local state, which survives every super step call.

5 Experimental Evaluation

Benchmark data were collected with the provided scripts in the `benchmark` folder (see section 5.1), while the data were gathered and displayed in section 5.5.
The test have been made on the *Xeon Phi* and the results shown refer exclusively to data collected on that machine.
The project was compiled with `g++` version `7.4.0` provided on the machine, and with the options provided in the `makefile`.
The program can be compiled with different preprocessor `define` in order to change a bit the behaviour of the programs. The possibilities are listed below (and are used in the `makefile` for various rules):

- `DEBUG`: (used in rule `make debug`) it activates the flag `-g` useful for debugging with `gdb`, and it checks that the output vector is equal to the one computed using `std::sort`.
- `BENCHMARK`: (used in rule `make benchmark`) it skips computing the `std::sort`, and only sorts the input array via the *Tiskin* algorithm.
- `TSEQ`: (used in rule `make tseq`) this signals that the user wants to collect some partial execution data (like the synchronization time, the super step time and so on). In this case the final time is higher due to the collection of partial times overhead.

5.1 Benchmark Folder

This folder is organized as follows:

- `benchmark.sh`: This bash script automates the benchmarking process. It starts the program with different parallelism degrees and different input vector sizes. It executes every combination of n (size of input vector) and nw 3 times in order to then compute an average of each time. The analysis of its output is delayed to section 5.3.
- `gprof-helper.c`: This file is needed in order to allow `gprof` profiling for multithreaded applications. It provides wrappers for the `pthread` creation.
- `profiler.sh`: This bash script automates the profiling process. At the end of the script execution, it dumps the profiling output in a file, by starting the program with various parallelism degrees. The analysis of its output is delayed to section 5.4.
- `tseq.sh`: This bash script collects some times from the program execution. In particular, it collects, for each worker, the *super step* time, the *barrier* synchronization delay and something similar. Also in this case, times are collected with various parallelism degrees. The analysis of its output is delayed to section 5.2.

All the cited scripts collect data and dump them in various files, that go in the folder `benchmark/data/`, which gets generated by those scripts.

5.2 Collection of Partial Times

In table 1 some partial times collected are shown. All the times are expressed in microseconds and are the result of an average between multiple runs. All the runs have been executed with 3 000 000 items ($2^{20} < 3M < 2^{21}$) as an input size.
The labels shown are described below:

- *ss computation*: it is an average of the time spent by a generic super step (average of the three) in computation phase.
- *ss communication*: same thing but related to communications.

- *barrier sync*: time spent by a generic super step waiting for all the workers to end their local computation and their communication. This is the value with the highest *variance*.
- *end process*: time spent by a generic worker waiting for the final synchronization, needed to assert the validity of the global continuation clause.
- *whole ss*: time spent by a generic worker in a generic super step (both communication and computation phase, but no synchronization).

phase \ *nw*	1	2	4	8	16	32	64	128	256
barrier sync	9	54	25	18	10	8	10	15	95
communication	0	11	10	7	5	7	17	7	216
computation	300	243	86	44	24	15	17	42	167
whole ss	293	250	93	49	28	20	29	47	316
end process	0	0	1	1	1	3	7	16	46

Table 1: Collections of partial times (in ms)

5.3 Benchmark

The raw data collected through the `benchmark.sh` script are shown in table 2. The time is shown in milliseconds, and it is the result of an average of multiple runs. The last column refers to the times collected using `std::sort`.

n \ *nw*	1	2	4	8	16	32	64	128	256	`std::sort`
2^{20}	294	306	146	86	64	66	113	336	2 684	186
2^{21}	595	645	258	161	112	96	144	343	2 675	392
2^{22}	1 219	1 417	530	303	203	152	200	381	2 643	823
2^{23}	2 494	2 787	1 060	586	362	280	295	722	2 211	1 705
2^{24}	5 175	5 781	2 431	1 186	712	454	429	923	1 539	3 577
2^{25}	10 751	12 309	4 867	2 575	1 393	851	770	1 198	1 797	7 460
2^{26}	22 390	25 220	11 353	5 191	3 245	1 665	1 224	1 543	4 016	15 579
2^{27}	48 066	56 732	23 548	12 145	6 788	4 928	2 191	2 132	5 041	30 804

Table 2: Time collected and averaged (in ms)

From a quick view, it is possible to notice that for a parallelism degree of 1 and 2, the running time is comparable. This seems to be due to the fact (according to the data collected with script `tseq.sh`) that, while super step 1 correctly "halves" its completion time on average, the third one increases a lot in running time. This is normal since super step 3 with 1 worker doesn't require queues and message passing, while the same cannot be said for 2 workers.
In addition to this, it is also possible to notice that (as it should be safe to assume), increasing the input size, the program scales better.
The program seems to scale well up until nw = 32 (with the exception of nw = 2).
When nw = 64, the program scales only for bigger input sizes (and by extension it is safe to assume that the same behaviour applies to nw = 128). For nw = 256 instead the overhead is huge.
Some plots related to those data are shown in section 5.5.

5.4 Profiling

According to `gprof`, the most time is spent iterating between elements of a vector. This is also due to the fact that a lot of time is spent in *generating* the input vector, and as the size grows of the input grows this time gets longer and longer.
That said, the most time consuming operations used by the workers, excluding the time spent iterating between elements, which is hard to monitor, are the *queue* related ones. Around 2% of the time is spent either *pushing* or *popping* from queues.
The longest super step is super step 2, which accounts for more than double the time spent in super steps 1 and 3.
The time spent locking and synchronizing around a barrier instead results *negligible*.
In order to execute again this evaluation, use the script `profile.sh`.

5.5 Plots

In this section some plots related to data generated with the script `benchmark.sh` are shown. Data are usually shown for the highest input size (2^{27}) and for an average on the used input sizes (for each $2^i, i \in [20, 27]$).

5.5.1 Completion Time

In figure 2 the completion time is shown on the left for an input size of 2^{27}, while on the right the average is shown. The crossing line is the time spent with `std::sort`.

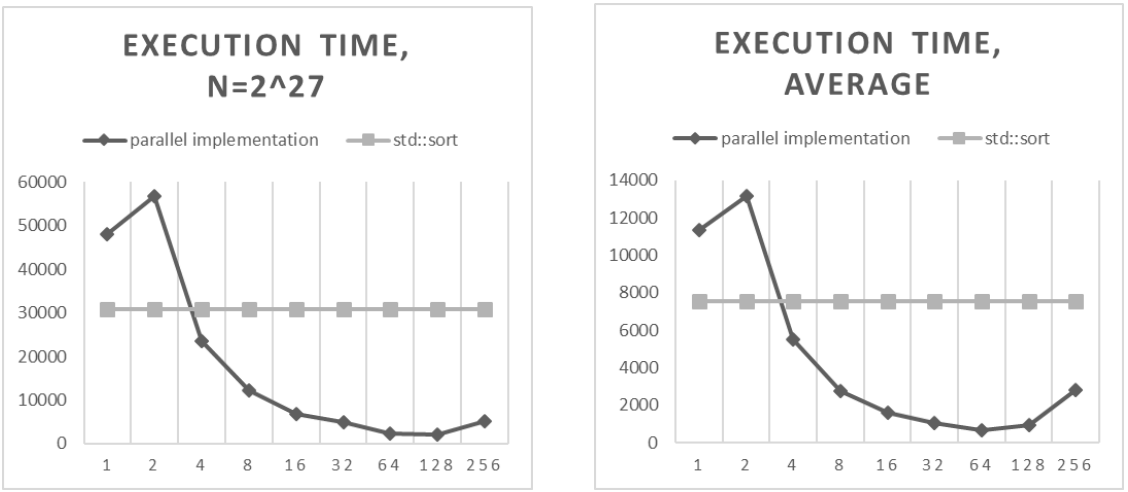

Figure 2: Completion time with max input size vs the average. (in µs)

5.5.2 Speedup

The *speedup* is computed as $s(p) = T_{seq}/T_{par}(p)$. The best sequential time T_{seq} taken as reference is the one obtained using `std::sort`.

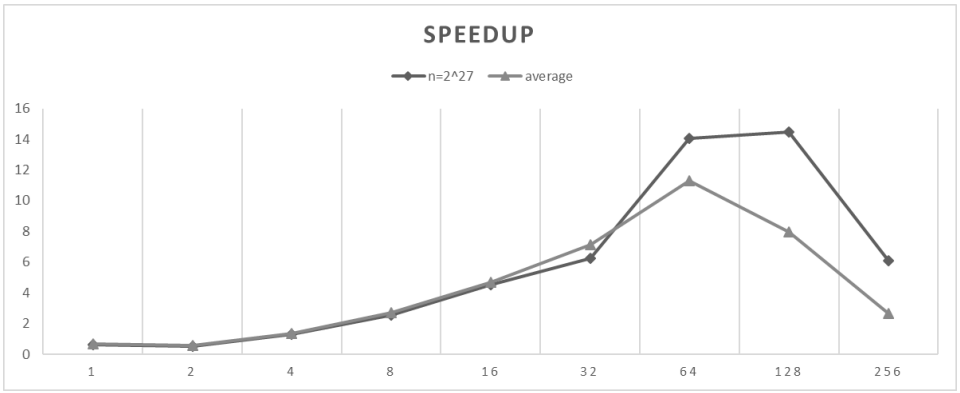

Figure 3: Speedup ($s(p) = T_{seq}/T_{par}(p)$)

5.5.3 Scalability

The *scalability* is computed as $scalab(p) = T_{par}(1)/T_{par}(p)$

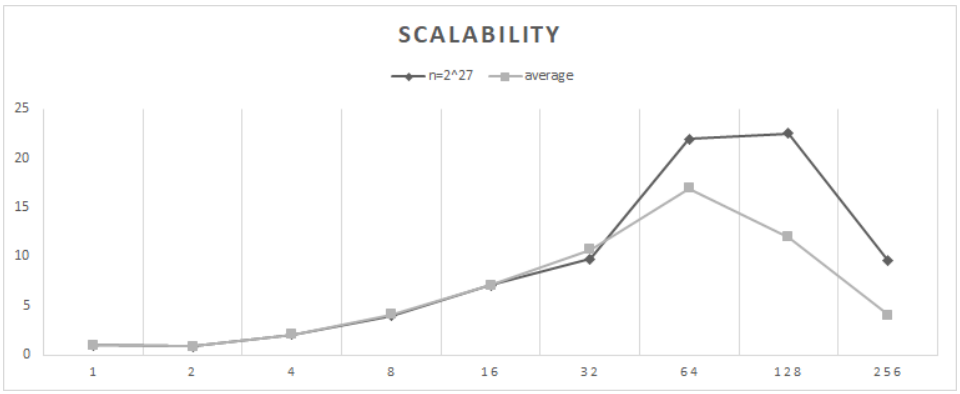

Figure 4: Scalability ($scalab(p) = T_{par}(1)/T_{par}(p)$)

5.5.4 Efficiency

The *efficiency* is computed as $\epsilon(p) = T_{seq}/(pT_{par}(p))$

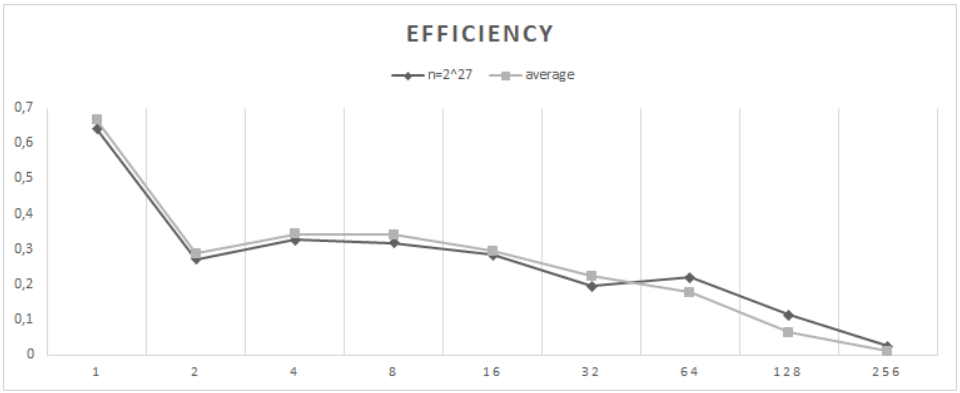

Figure 5: Efficiency ($\epsilon(p) = T_{seq}/(pT_{par}(p))$)

6 Conclusions