TRINITY COLLEGE DUBLIN, SCHOOL OF MATHEMATICS

5691 Seminar Report # 2
A Framework for an Automatic Hybrid MPI+OpenMP code generation

Gustavo Ramirez

April 15, 2017

1 INTRODUCTION: BSP++ AND THE THE FRAMEWORK BUILT AROUND IT

In a paper from Hamidouche *et al.* [2], the BSP++ library is developed. It basically consists in a tool for generating parallel code from a sequential version in an automated way.
The automatic creation of parallel code from a sequential one is, in general, a complicated and very interesting problem. Clearly, how difficult the implementation of the automatic translator will be depends on the initial assumptions, including the kind of problems that will be able to be automatically driven into a parallel version, and also the kind of hardware involved.
The framework discussed here was developed in an article by Hamidouche *et al.* [1], and makes use of the BSP++ library; it is a system wrapping the BSP++ library, and adding the rest of layers necessary for making a completely automated system suitable for the creation of parallel code from its serial initial implementation.
There are multiple reasons for using such an automation tool. For instance, writing good parallel code requires both expertise and performance analysis, the latter being very time consuming, and the former implying a long time of hard work; even then, hybrid code can end up being inefficient due to many possible issues, like different types of overheads (at the MPI level, OpenMP level, etc.), load balancing, etc. Therefoce, one of the main motivations of such a tool is not dealing with all of that.
In short, the main goal of the framework described here is to produce efficient hybrid code for multi-core architectures from the source of a sequential function.

2 THEORY: BSP AND SPMD

BSP stands for Bulk Synchronous Parallel; it is synchronous in the sense that the work is divided in supersteps, and the estimations for execution times are then taken on each superstep. A diagram of one superstep in the BSP model can be seen in figure 2.1.
As seen from figure 2.1, the BSP model consists of two overlaped stages, i.e. the computate stage and the communicate stage, the latter followed by a waiting stage, where basically calls to full synchronizations are made. A simple linear mathematical estimation of the execution time for such a superstep is of the form displayed in equation 2.1, where:

- δ: execution time for one superstep.

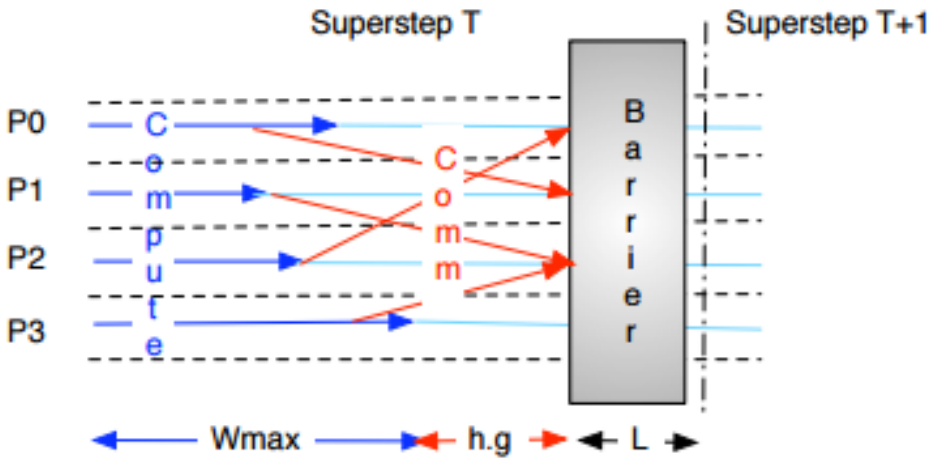

Figure 2.1: one superstep in the BSP model. Image taken from [1].

- W_{max}: from the execution time for all the processes on their computations stage within one superstep, the maximum of all of those values.
- h: maximal amount of data sent or received by one processor.
- g: communication speed.
- L: synchronization duration.

$$\delta = W_{max} + h \cdot g + L \tag{2.1}$$

The BSP model is subdivided in two submodels:

1. machine model: this model is composed of a set of processors linked through a communication medium supporting point-to-point communications and synchronizations. Some parameters associated to this submodel:
   - P: number of processors.
   - r: speed, in FLOPS.
   - g: communication speed.
   - L: synchronization duration.
2. programming model: as displayed in figure 2.1.

Some cons of the BSP model:

- global synchronizations can cost a lot of execution time; this can become dominant for large parallel machines, where the number of communicating processors is too large. To reduce this problem, a possibility is launching threads within the processes, so that the amount of communications are reduced.

and some pros:

- it has a very simple analytical model, which implies a simple estimation of the execution time.
- there is a performance improvement; the "best" number of processors and threads are found.

As mentioned in the last previous point, the "best" numbers of processors and threads are found. This basically implies an optimization on the execution time with some set of parameters, like communication speeds, finding then the best hybrid (MPI+OpenMP) configuration.
Going even further on the simplifications, a **SPMD** (Single Program Multiple Data) model is implied; this model implies that tasks are split up and run simultaneously on multiple processors with different input in order to obtain results faster. The SPMD assumption works very well when the OpenMP level is included, as each threads works on its own data and in a complete parallel way; SPMD implies, partly, not inserting serial sections between two parallel (OpenMP) sections. Using other mode different from SPMD might imply high thread creation overhead, and serial sections may become bottlenecks.
SPMD has several pros, such as the performance prediction becoming easier (there is a splitting into computation and communication times), false sharing is avoided, and load balancing is good, as all threads execute the same program.

3 Implementation: analyzer, searcher and generator

3.1 Implementation specs

The most significant assumptions over the system to be implemented are:

- it uses the BSP cost model to estimate execution time

- the implementation finds the best configuration (number MPI processes and OpenMP threads)
- generates hybrid code automatically

There are two inputs from the user: user source file containing sequential code and an XML describing some characteristics of the parallel code to be generated. The main output is hybrid MPI+OpenMP code.

3.2 Analyzer

This stage generates a numerical formula for execution time (with data size as parameter). It is used to estimate T_{comm}: it counts the number of cycles needed to execute a sequential function. The way it performs such a calculation, is with the use of the compiler Clang[1] (with all optimizations: -O3) to generate bytecode, which is then used as input for LLVM[2] (which counts the number of cycles in the bytecode).
This is stage is further subdivided into:

- *processor model* (no cache misses): the total execution time is calculated as $\sum_i C_i$, with C_i the hardware cost of each operation (e.g. load, store, etc.) i.
- *cache model* (for cache misses): total cost equals $\sum_i^{levels}(M_i \cdot PEN_i)$, with the penalty PEN_i given in nuber of cycles, and the number of cache misses M_i is as given in equation 3.1, where j represents instructions, β_j represents the amount of memory and γ_j the number of references associated to that amount of memory for that instruction # j.

$$M_i = \frac{1}{CAPACITY_i} \sum_j (\gamma_j \beta_j) \tag{3.1}$$

Due to the simplicity of the BSP model, then the execution time for all levels (MPI, OpenMP, and more levels like CUDA is included) is as in equation 3.2.

$$T_{comm,k} = h_k \cdot g_k(P_k) + L_k(P_k),\ k \in \{MPI, OpenMP\} \tag{3.2}$$

At this point is when the *system profile* comes in, which combines many benchmarks and sphinx[3] to find $g_k(P_k)$ and $L_k(P_k)$.

[1] see [3].
[2] see [4].
[3] see [5]; sphinx is a tool for running performance tests of MPI, Pthreads and OpenMP.

Finally, the total execution time goes as a summation over all execution times, i.e. $T_{comm} = \sum_k T_{comm,k}$.

3.3 SEARCHER

This stage uses the previous formula and info from the *system profile* to estimate cost for all possible configurations. The system profile retrieves execution time for basic operations, obtained either from the processor's manual and design specs, or by benchmarking.
It uses the XML and the data from the analyzer, in order to be able to find the best combination number of MPI processes and OpenMP threads.
Important factors taken into account by this stage are:

- number of MPI processes
- number of threads per process
- data size of the application

The searcher then plots a graph, as in figure 3.1.
Each vertex is represented as $V(S_i, n, m, o)$, where S_i represents the i-th superstep, n the number of nodes, m the number of processes and o the number of threads, and an edge is a transition between two vertices, i.e. $V(S_i, n, m, o) \rightarrow V(S_{i+1}, n, m, o')$; the number of nodes and processes are fixed as MPI restrictions. After the graph is fully created, then Dijkstra's algorithm[4] is used to find the shortest path, which implies shortest execution time.

3.4 GENERATOR: BSP++

The third and final stage uses the configuration chosen through Dijkstra's algorithm and generates the corresponding hybrid code, with the use of the BSP++ library.
This part is based in a previous paper from the authors[5]. One of the most important points about the BSP++ library is that it implies a hybrid structure of the form displayed in figure 3.2; as can be seen in figure 3.2, the implementation is simply an insertion of OpenMP code in the computational MPI parts.

[4] see [6].
[5] see [2].

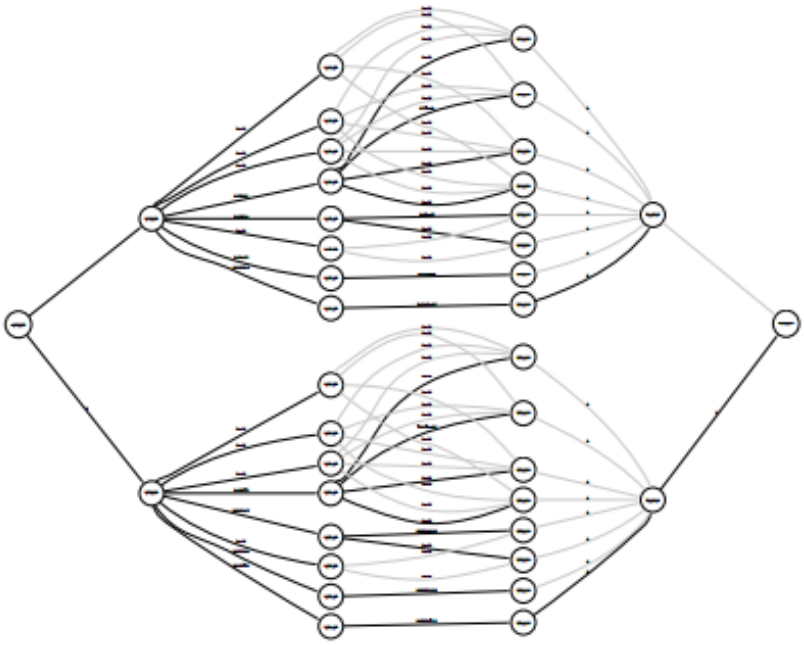

Figure 3.1: graph created by the searcher stage before finding the best execution time. Image taken from [1].

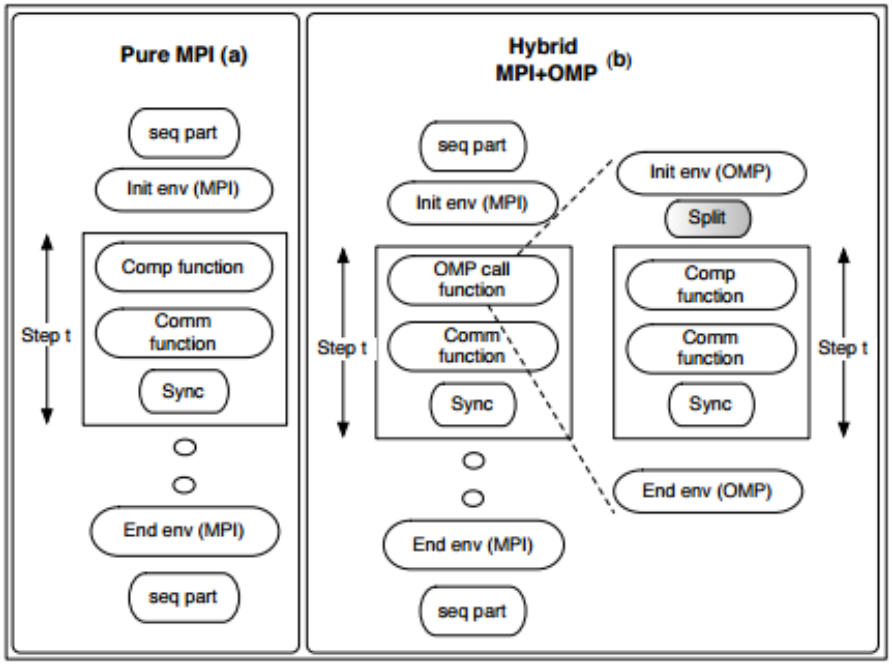

Figure 3.2: hybrid structure on which the generator stage is based. Image taken from [1].

4 Benchmarks, results and conclusions

4.1 Benchmarks

On testing the framework developed, four different benchmarks were used:

- inner product.
- vector-matrix multiplication.
- matrix-matrix multiplication.
- parallel sorting by regular sampling algorithm.

Here, the first of those benchmarks is discussed; also, the hardware used consisted of 4 nodes, each node being a bi-processor with a bi-core and 2.6 GHz, 4 GB RAM and 2 MB of L2 cache.

4.2 Results

As can be seen from figure 4.1, for small data sizes the computation time is not larg enough, so that communication times represent a large portion of the total execution time when the number of processes is large (like in pure MPI); therefore, for small amounts of data, putting only threads to do all the computations is ideal, as the overhead of communicating between processes becomes important, and the double overhead of having MPI processes launching threads becomes important as well.
On the other hand, when the amount of data is large, as in figure 4.2, having as many cores as possible handling each of them one thread, is ideal; then, as seen in that figure, the fully hybrid version becomes the best option, as it leads to the lowest execution time.

4.3 Conclusions

- The integration of more layers (e.g. CUDA) in the framework becomes relatively simple, as the assumptions imply BSP and SPMD.
- The framework detects recursiveness automatically from the sequential code, and implements the parallel version taken this feature into account.
- The described framework selects from pure MPI, pure OpenMP and hybrid MPI+OpenMP, and chooses the best hardware environment for the minimization of the total execution time.

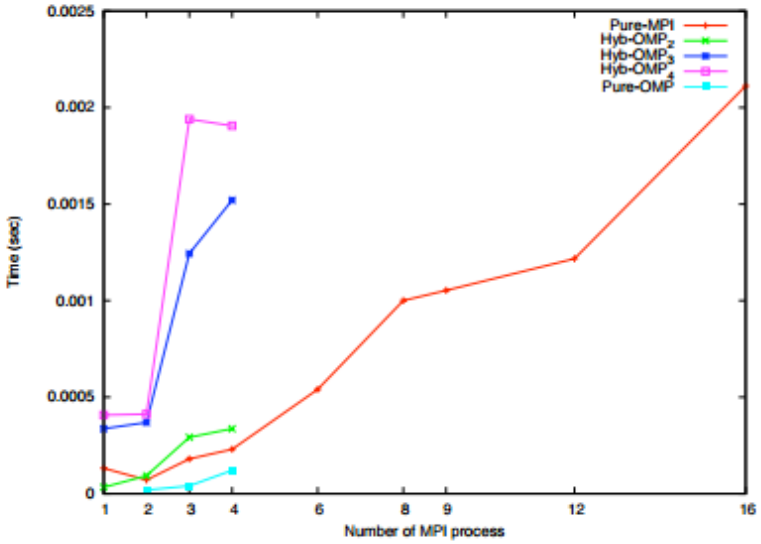

Figure 4.1: execution time for all possible configurations for the inner product program with a small data size (64 k). Image taken from [1].

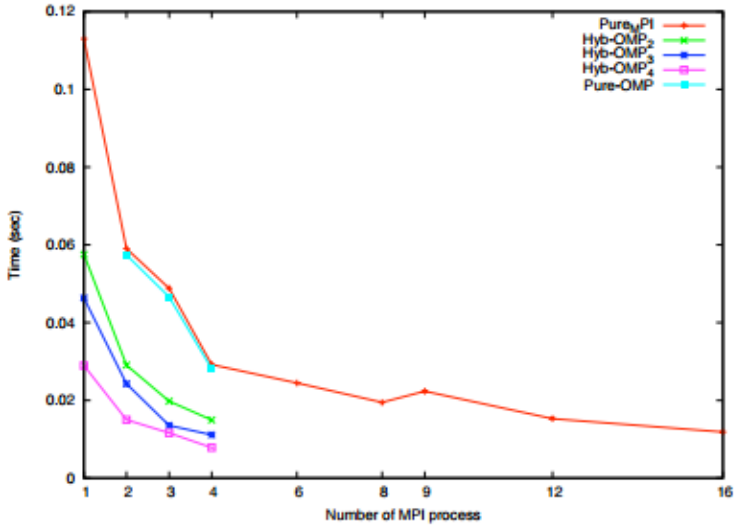

Figure 4.2: execution time for all possible configurations for the inner product program with a large data size (128 M). Image taken from [1].

REFERENCES

[1] Khaled Hamidouche, Joel Falcou, Daniel Etiemble *A Framework for an Automatic Hybrid MPI+OpenMP code generation*. Proceedings of the 19th High Performance Computing Symposia (HPC '11) (2011)

[2] Khaled Hamidouche, Joel Falcou, Daniel Etiemble *Hybrid bulk synchronous parallelism library for clustered smp architectures*. Proceedings of the fourth international workshop on High-level parallel programming and applications (HLPP '10) (2010)

[3] https://clang.llvm.org/

[4] http://llvm.org/

[5] http://www.sphinx-doc.org/en/stable/

[6] http://www.personal.kent.edu/ rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/ dijkstraAlgor.htm