A Framework for an Automatic Hybrid MPI+OpenMP code generation

Khaled Hamidouche, Joel Falcou, Daniel Etiemble
Laboratoire de Recherche en Informatique
University Paris-Sud XI, Orsay, France
hamidou,joel.falcou,de@lri.fr

Keywords: Generic Programming, Bulk Synchronous Parallelism, Performance Prediction, OpenMP, MPI, MPI+OpenMP

Abstract

Clusters of symmetric multiprocessors (SMPs) are the most currently used architecture for large scale applications and combining MPI and OpenMP models is regarded as a suitable programming model for such architectures. But writing efficient MPI+OpenMP programs requires expertise and performance analysis to determine the best number of processes and threads for the optimal execution for a given application on a given platform.

To solve these problems, we propose a framework for the development of hybrid MPI+OpenMP programs. This paper provides the following contributions: (i) A compiler analyser that estimates the computing time of a sequential function. (ii) A code generator tool for generating hybrid code based on the compiler analyser and a simple analytical parallel performance prediction model to estimate the execution time of an hybrid program. (iii) An evaluation of the accuracy of the framework and its usability on several benchmarks.

1. INTRODUCTION

Clusters of symmetric multiprocessors (SMP) are the most cost-effective solution for the large scale applications. In the Top500 list of supercomputers, most of them (if not all) are clusters of SMPs. In this context, using a hybrid MPI+OpenMP approach is regarded as a suitable programming model for such architectures. Some works [8, 13] have presented the performance improvement when using MPI+OpenMP. On the other hand, there are also significant reports of poor hybrid performance [7, 10] or minor benefits when adding OpenMP to an MPI program [5, 18]. The factors that impact the performance of this approach are numerous, complex and interrelated:

- MPI communications efficiency: performance of the MPI communication constructs (point to point, broadcast, all-to-all, ...), message sizes, interconnection latencies and bandwidths are key factors. applications related problems such as MPI routine types (point to point, collective), message size and network related problems such as network connection and bandwidth.
- OpenMP parallelization efficiency: using critical section primitives, the overhead of OpenMP threads management and false sharing can reduce the performances.
- MPI and OpenMP interaction: load balancing issues and idle thread inside the MPI communication part reduce the parallel efficiency.

To obtain an efficient hybrid program one must determine the number of MPI processes and OpenMP threads to be launched on a given platform for a given data size. With N nodes, each of them having p cores, it is customary launch N processes with p threads per process. This choice raises two potential sources of performance loss. First, as the volume of interprocessor communication increases with the number of processes, the consequent overhead could become a disadvantage. Second, concurrency between threads (data and/or explicit synchronization) is also a factor that limits the scalability, thereby leading to performance stagnation or slowdown. In addition to this performance aspect, there is a technical effort to derive and deploy hybrid programs. to solve these problems, developers need solutions in terms of effective automatic parallelization tools and libraries.

The main contribution of this work is the design and development of a new framework that is able to produce an efficient hybrid code for multi-core architectures from the source code of a sequential function. Some consequences of this contribution are:

- The developers can generate an efficient hybrid code without learning new materials, just by using a list of sequential functions.
- The integration of the existing legacy codes, such as the codes using the SPMD model, can be ported without significant loss of efficiency. It is the case for many sequential or MPI codes.

This paper is organized as follows: Section 2 presents the related works. Section 3 describes the framework and its architecture, The experimental results on various HPC algorithms are given in Section 4. Finally, Section 5 sums up our contributions and opens on future works.

2. RELATED WORK

2.1. Existing tools

The MPI+OpenMP programming model has been extensively discussed in the literature, including source analysers, performance-analysis and parallel computing models.

In [9] the authors implemented a source code analyser to predict the performance of MPI applications. The compiler tool parses the source code to extract informations on the MPI routines and the arithmetic operations. Within the PERC project [20], the authors combine two profiles: the *Machine profile* and an *application signature* to estimate the performances of sequential and/or MPI programs. The combined process is called the *convolution*, which is a "mapping" of the application signature within the machine profile.

The *EXPERT* tool [23] performance-analysis environment is able to automatically detect the performance issues in event traces of MPI, OpenMP or hybrid MPI+OpenMP applications running on parallel computers with *SMP* nodes. The EXPERT analysis process is composed of two parts: a semi-automatic multi-level instrumentation of the user application followed by an automatic analysis of the generated performance data.

In [3], the authors developed a performance model of hybrid MPI+OpenMP applications. The framework uses their own compiler named *OpenUH*. In [17],the paper presents a compile time cost model for OpenMP programs: the model estimates the CPU processing time,the memory access time and also the overhead of OpenMP pragmas.

In [19], the paper presents a tool for the automatic generation of hybrid code . This tool uses LLC, a high level parallel language. Different directives have been designed and implemented in LLC to support the LLCOMP [19] annotation directives. The hybrid MPI+OpenMP code is generated using the LLCOMP directives.

On the other hand, there are many performance models for parallel programming such as LogP [11], LogGP [4] and PLogP [15]. LogP predicts the communication performance of small messages. LogGP is an extension of the LogP model that additionally consider large messages by introducing the gap per byte parameter. PLogP is another extension of the LogP model that considers the cost of data copies to and from the network interfaces. Our code generator is based on the BSP - **Bulk Synchronize Parallel**- model

2.2. The BSP model

The Bulk Synchronous Parallel Model (BSP) was introduced by Leslie G. Valiant [21] as a bridge between the hardware and the software to simplify the development of parallel algorithms. Classically, the BSP model is defined by three components:

- **A machine model** that describes a parallel machine as a set of processors linked through a communication medium supporting point-to-point communications and synchronizations. Such machines are then characterized by a set of experimental parameters[6]: P, the number of processors; r, the processor speed in FLOPS; g, the communication speed and L, the synchronization duration.
- **A programming model** that describes how a parallel program is structured (Figure 1). A BSP program consists of a sequence of super-steps in which each process performs local computations followed by communications. When all processes reach the synchronization barrier, the next super-step begins.

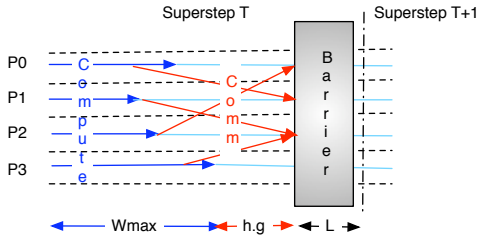

Figure 1. An overview of the BSP Programming Model

- **A cost model** that estimates the duration δ of a super-step as a simple function combining the machine parameters (L and g), the maximal amount of data that is sent or received by one processor (h) and the maximal workload of all the processors (W_{max}) . For any given super-step, we have:

$$\delta = W_{max} + h.g + L$$

One common objection against BSP is the cost of global synchronizations that can become dominant for large parallel machines. This cost can be reduced by using the efficient communications between SMP cores using OpenMP to minimize the synchronization overhead and to perform fast communications. The advantages of the BSP model for hybrid systems are: 1) As it is based on a simple analytical model, the BSP cost model facilitates the execution time estimation. 2) As the underlying model of BSP is SPMD, the use of the OpenMP SPMD model on shared memory architecture with the OpenMP paradigm provides a performance improvement compared to MPI and fine-grain OpenMP [16, 22].

The fine-grain OpenMP model within the hybrid parallelization is very complicated to predict as it uses many directives and critical clauses such as *single*, *master*, *threadprivate* ...etc. We propose to use the SPMD style that is far more simple. It allows :

- The performance prediction, as the execution time can be split into *Computation time* and *Communication time*.
- It avoids the **false sharing** issues, as we use private vectors on each thread.
- The load balancing is good, as all threads execute the same program.

3. METHODOLOGY

To facilitate the development of efficient programs on clusters of SMP architectures, we propose a framework that:

- Uses the BSP cost model to estimate the execution time.
- Finds the best configuration (number of MPI process and the number of OpenMP threads) over all valid configurations.
- Generates the corresponding hybrid code.

Our framework tool follows the ideas of [3] and the PERCS project [20], but uses an analyser instead of an application signature to estimate the execution time of the program on the architecture.

3.1. Global view

The framework tool consists of three modules: *Analyzer*, *Searcher* and *Generator*. It takes as inputs the user source file containing the sequential functions and the XML file that describes the parallel code. It produces the hybrid MPI+OpenMP program as an output.

Our framework tool works as follow: first the *Analyzer* retrieves the estimate computation time for each function in the user source file (1). Since the data size is known only at runtime in most case, the *Analyzer* generates a numerical formula with the data size as a parameter. Then the *Searcher*, uses this formula and the informations extracted from the *System profile* (2). These informations are derived from the runtime benchmarks and the communication patterns extracted from the user *XML* file (1). The *Searcher* computes the estimated cost of all possible configurations. After choosing the best configuration, the *Generator* generates the corresponding code using a customized library (3). The framework overall architecture is depicted in Figure 2.

In the following section we describe each module of the framework and the global work flow. The "*inner product*" is used as an example to show the effect of each step.

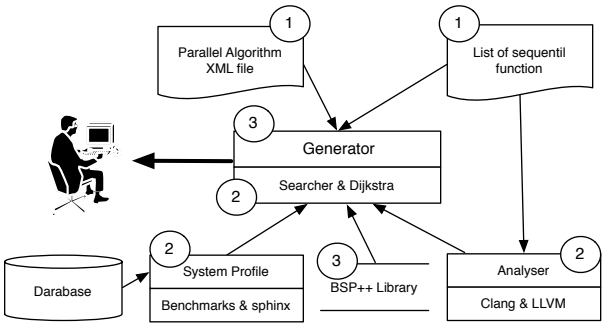

Figure 2. Overall architecture of the framework

3.2. Analyzer

The *Analyser* module estimates the cost model by combining the static analysis and measures of runtime benchmarks. The cost model of a step under the hybrid BSP model is

$$\delta = T_{comp} + T_{comm} + T_{lc}$$

T_{lc} (Time Level Changing) corresponds to the overhead for managing the OpenMP threads due to the switch between the MPI level and OpenMP level. T_{comp} represents $W_{i_{max}}$ in the original BSP analytic model and T_{comm} corresponds to $h * g+L$. The *Analyzer* module involves two models: the Computation Model and the Communication Model.

3.2.1. The Computation Model

This model is mainly used to estimate T_{comp} by counting the number of cycles needed to execute a sequential function. A source parser is not accurate as it cannot detect the compiler optimizations. A more accurate estimation consists in parsing the code generated by a compiler with all optimization (-O3). To do so, we use Clang [1] to generate the byte-code of the user source code. Then we implement a new pass in the LLVM [1] compiler to count the number of cycles in the byte-code. For each function in the byte-code, the parser counts the number of instructions in each basic block and in each loop block. Using a database mapping the number of cycles of each instruction in a basic block, the parser generates the *Numerical* formula for each function.

This model itself is decomposed into two sub-models: *Processor Model* and *Cache Model*.

Processor Model: The sub-model for the CPU estimates the number of cycles spent doing computation. It counts the number of operations executed in the CPU functional units, including load and store operations assuming there is no cache misses. In addition, it considers intrinsic functions like SQRT as basic operations [9]. The model takes into account the difference between FP and ALU operations. The model can be expressed as the sum of *the hardware cost* (C_i) for each operation *i*. The hardware cost C_i is derived from the *system Profile*, which can be obtained either from the processor's manual and the design specification, or by benchmarking [9].

Listing 1 depicts the user source file containing the sequential functions for the inner product program.

Listing 1. BSP++ list of sequential functions : inner product

```
1 double inner-product( vector<double> const& in)
2 {
3 return std::inner_product(in.begin(),in.end(),in.begin()
      ,0);
4 }
5 double Global-sum( vector<double> const& in)
6 {
7  return std::accumulate(in.begin(),in.begin(),0);
8 }
```

For this byte-code, the parser counts 1 cycle for each *add* instruction at line 4 and line 8 and for the *icmp* instruction. It counts 3 cycles for the load instruction at line 7. Then we get 6 cycles for each iteration of the loop and there are *N* iterations. The parser counts also 7 cycles outside the loop block corresponding to the initialization, and produces this numerical formula:

$$Global-sum : 7+6*N$$

Listing 2 represents the byte-code corresponding to the loop in the $Global-sum$ function generated by Clang.

Listing 2. BSP++ Byte-code: Global-sum function

```
for.body.i:     ; preds = %entry, %for.body.i
  %indvar.i = phi i64 [ %tmp, %for.body.i ], [ 0, %entry ]
  %__init.addr.010.i = phi i32 [ %add.i, %for.body.i ], [ 0, %entry ] ;
  %tmp = add i64 %indvar.i, 1                                         ;
  %ptrincdec.i.i = getelementptr i32* %tmp3.i.i.i, i64 %tmp ;
  %__first.0.09.i = getelementptr i32* %tmp3.i.i.i, i64 %indvar.i ;
  %tmp2.i6 = load i32* %__first.0.09.i;
  %add.i = add nsw i32 %tmp2.i6, %__init.addr.010.i ;
  %cmp.i.i = icmp eq i32* %ptrincdec.i.i, %tmp3.i.i.i8 ;
  br i1 %cmp.i.i, label %_ZStaccumulate, label %for.body.i
```

Cache Model: This model estimates the time spent accessing data in the memory hierarchy [9]. As mentioned before, for the processor model, we assume all memory reference are cache hits in the L1 cache. However the execution time of a program depends on the number of data misses on the different cache levels.
The Cache-Model can be expressed as follows:

$$Cost_{cache} = \sum_{i}^{levels} (M_i * Penalty_i)$$

where M_i is the number of cache misses at the i^{th} level of the memory hierarchy. $Penalty_i$ is expressed on cycles and it represents the number of cycles needed to access the i^{th} level.

The parser accumulates the footprint of each instruction that represents the number of data references multiplied by the size of the elements (β). If the number exceeds the cache capacity, then we assume a cache miss on this level. Thus, M_i is :

$$M_i = (\sum_{j}^{inst} (ref * \beta))/Capacity_i$$

This analytical cache model is not accurate, but we don't need more accuracy because : 1) the estimated computation time is just used to decide how many processors must be used and in which mode (MPI or Hybrid) by computing the ratio $\frac{parallel\ computation}{parallel\ communication\ time}$. 2) as both OpenMP and MPI use the same SPMD style they have the same memory access pattern that leads to the same cache behaviours.

3.2.2. Communication Model

Our communication model requires the values of g and L. As they depend on the *level* of the hybrid BSP machine (MPI or OpenMP) and the number of processors [14], we express them using the level and the number of processors at each level: $g_k(P_k)$ and $L_k(P_k)$ where k represents the level and P_k the number of processors for the k^{th} level. Then we have :

$$T_{comm} = h_k * g_k(P_k) + L_k(P_k)$$

The $system\ profile$ is used to find these values. The profile combines many benchmarks and the *Sphinx* tool [2] to determine the accurate values. As the program uses a BSP-based library, we need to evaluate the overhead of a subset of the MPI routines and the OpenMP directives (*MPIAlltoAll*, *MPIAllGather*, *OMP PARALLEL SECTION* and the *Barrier Synchronization* for both MPI and OpenMP). The current version of *Sphinx* was sufficient to get all the results. The Communication Time is the sum of sub-communication times at each level and the sub-communication time is:

$$T_{SubComm_i} = h_i * g_i(P_i) + L_i(P_i) + T_{lci}(N)$$

3.3. Searcher

After computing the cost for the user algorithm we use it to find the best configuration. The performance of a hybrid MPI+OpenMP program depends on the number of MPI processes, the number of OpenMP threads per MPI process, and the data size of the application. Finding these numbers for each step of the program is a critical task. One way to find these values is to solve the partial derivative equations of the analytical cost model: this is a complex task as we must find the global minimum for each variable for the entire program and the corresponding values for each step. A simple solution is to make an exhaustive search on the entire space of configurations. To do so, we build a graph and explore all valid configurations.We denote $G = (V,E)$ the *Direct Acyclic Graph* (*DAG*) where V is the set of vertexes. Each vertex is a configuration of execution for a given step with the corresponding mode (MPI/OpenMP). E represents the set of edges. A vertex is a quadruplet $V(S(n\{m\{o\}\}))$ where S is the step to be launched, n the number of physical nodes that are used, m the number of MPI processes launched per node and o the number of OpenMP threads per MPI process.

The weight of an edge $V(S_i(n\{m\{o\}\}))- > V'(S_{i+1}(n\{m\{o'\}\}))$ is the cost of executing S_{i+1} with the configuration $S_{i+1}(n\{m\{o'\}\})$ knowing that S_i has been executed with the configuration $S_i(n\{m\{o\}\})$. The cost of the edge depends on the configuration in which the step S_{i+1} will be executed, because of the global synchronization with MPI, the OpenMP threads management and the data split over thread overheads. The estimated cost of an edge is carried out by the *Analyzer* module and the communication pattern extracted from the *XML* file.

The Graph G is then hierarchically built. First we choose the number of physical nodes to be used. There is a sub-graph for each number. Then, inside each sub-graph, we choose the number of MPI processes to be launched and for each MPI process the number of OpenMP threads. As the cluster is homogeneous, and we use the SPMD model, all nodes may utilize the same configuration (with 2 nodes, the same configuration is running on each node). Thus, the number of configurations can be expressed as:

$$V = q * (S * \alpha + 2) + 2$$

where q is the node number of the Cluster, S is the number of steps of the program and α is:

$$\alpha = \sum_{k=1}^{P} card\{(m,o) \in \mathbb{N}^2 : m * o = k : k \leq C, 1 \leq o \leq C\}$$

where C is the number of cores in a node. α represents the number of valid configurations in a node: the number of MPI processes $\times$ number of OpenMP threads is less or equal to the number of cores in the nodes.

The number of MPI processes or the number of nodes cannot be changed between steps. In other words, there is no edge between two sub-graphs and inside a sub-graph there is no edge between two configurations with different values of m (number of MPI processes).

Finally,we use the Dijkstra's shortest Path algorithmto find the fastest execution along the graph edge.

Listing 3. XML file for the BSP inner product program :

```
<program  name= inner-product>
 <step id =1>
  <Compute>
    <F_name> inner-product </F_name>
    <Arg> 6400000</Arg>
  </Compute>
  <Communicate>
    <Comm_type> Proj </Comm_type>
  </Communicate>
 </step>

 <step id=2>
  <Compute>
    <F_name> Global-sum </F_name>
      <Arg_Exp> NB_PROC </Arg_Exp>
  </Compute>
 </step>
</program>
```

The XML file (Listing 3) describing the Parallel algorithm of the inner product program involves two steps. During the first one, a local computation of a the "*partial*" result is done using the *inner* − *product* function, followed by a communication phase to gather all the partial results on all nodes. Afterwards, an accumulation of all the received partial results is implemented within the "*Global* − *sum*" function.

The shortest path of the generated graph for the BSP inner product program (figure 3) using the information of the BSP machine (2 nodes, 4 cores/node) extracted from the system profile, shows that the step 1 is executed with the hybrid MPI+OpenMP model using 2 nodes (2 MPI processes) with 4 OpenMP threads per node. However, as the second step requires fewer cycles (the data size is equal to the number of MPI processes), executing it on one core is the fastest solution because of the overhead to manage OpenMP threads and the synchronization.

3.4. Code generation with BSP++

After determining the Shortest Path, we generate the corresponding parallel code using our BSP++ library [14]. BSP++ is an object-oriented implementation of the functional BSML library[12]. To localize the source of potential parallelism, it

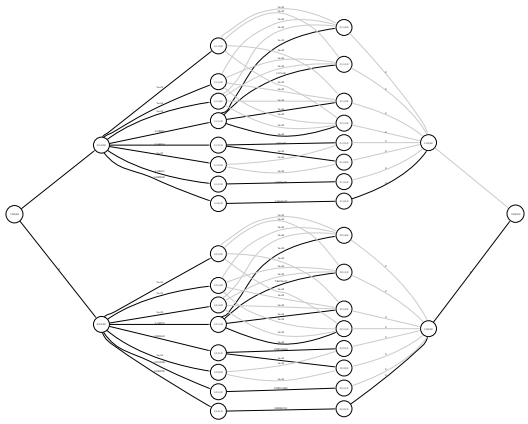

Figure 3. The Graph generated for the inner product example. Bold edges represent the Shortest Paths

provides a structured interface to the BSP model based on the notion of *paralleldatavector*. In this model, the user stores the distributed data in a specialized generic class called *par* that supports BSP style communication patterns.

In addition to C-style functions and classical C++ function objects, BSP++ also supports the two main implementations of lambda function in C++ : the *BOOST.PHOENIX* library and the $C++0x$ style lambda function.

The hybrid code is obtained by using an OpenMP enabled super-step function inside a MPI super-step computation phase as shown in Figure 4. Basically, part (a) shows the typical layout of an MPI superstep. To enable hybrid computation (figure 4 (b)), the computation phase of the MPI superstep is replaced by a call to an OpenMP BSP function. A *split* of MPI data into OpenMP private variables by passing a range of iterators may be needed.

However, with an hybrid configuration, for some algorithms, replacing the computation function by an OpenMP function including only the step, gives an incorrect hybrid code. To generate a correct hybrid code, the function must be replaced by the entire BSP program. To do so, when the *Searcher* and *Generator* detect the **recursive** call (i.e. the sequential user function of a step has the same name than the BSP program in the XML file), they respectively estimate, generate the OpenMP BSP function for this step including the entire BSP program.

Listings 4 and 5 show the hybrid code of the inner-product program generated by the *Generator* module using the BSP++ library. In listing 4, the OpenMP function shows an example of recursive call generation. Indeed, in the user XML file, the name of sequential function in step 1 is the same as the name of the BSP program, then the *Generator*, in a hybrid configuration, generates an OpenMP function including the entire BSP program (both steps) for the first step.

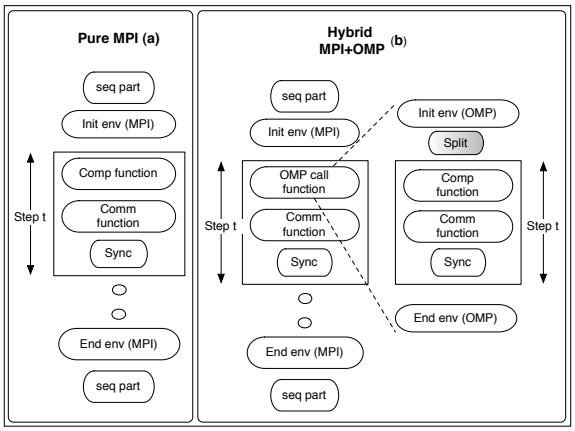

Figure 4. Parallelization with the hybrid model. The MPI computation phase is replaced by a call to a BSP function compiled in OpenMP mode.

Listing 4. BSP++ hybrid inner product: OpenMP function

```
double omp_inner_prod( vector<double> const& in, int argc,
      char ** argv)
{
  double value;
  BSP_HYB_START(argc, argv)
  {
    result_of_split< vector<double>,linear() > v = split(
        in);
    *r = inner-product (*v);
    result_of_proj<double> exch = proj(r);
    // step 2
    value = Global-sum( exch.begin() );
    synch();
  }
  BSP_HYB_END()
  return value;
}
```

Listing 5. BSP++ hybrid inner product: MPI part

```
int main (int argc, char** argv)
{
   par< vector<double> > data;
   par< double >         result;
   omp_set_num_threads(4);
   *result= omp_inner_prod( *in, argc, argv);
    result_of_proj<double> exch = proj(result);
   //  step 2
   *result = Global-sum ( exch.begin() );
   synch();
}
```

4. PERFORMANCE EVALUATION

In this section, we evaluate the performance of the solution that is provided by our tool according to the number of MPI processes and OpenMP threads. In other words, we compare the best solution generated by the tool and the execution times of all correct configurations.

The evaluation is based on four benchmarks: the inner product, the vector-matrix multiplication (**GMV**), the matrix-matrix multiplication (**GMM**) and the Parallel Sorting by Regular Sampling Algorithm (**PSRS**). Our experiments were conducted on a cluster located in the Bordeaux site of the GRID5000 platform. We used four nodes connected by a 2 x BCM5704 Gigabit Ethernet Network. Each node is a bi-processor bi-core 2.6-GHz AMD Opteron 2218 with 4-GB of RAM and a 2-MB L2 cache. The MPICH2.1.0.6 library was used. For each benchmark and each algorithm, we measured the median execution times of 100 runs.

4.1. Model accuracy

Table 1 reports the results of the communication performance predictions for the benchmarks with the hybrid implementation. The error ranging between 4% and 17% (except for the Inner Product benchmark) asserts the accuracy of the sub-model. Knowing that the standard deviation of sphinx is up to 150% [2], we believe that this error is acceptable. For the *innerprod* benchmark, the large difference between the estimated and the measured run-times (up to 40%) is due to the communication pattern. In this benchmark, each processor only sends *One* element to all other ones.

Table 1. Comparison of measured and predicted communication times on different benchmarks

Bench	*CPUs*	$Estim_{Time}$	$Meas_{Time}$	*Error%*
InProd	4	$5\ 10^{-5}$	$5.4\ 10^{-5}$	7
	8	$2.3\ 10^{-4}$	$3.9\ 10^{-4}$	40
	16	$3.4\ 10^{-4}$	$5.2\ 10^{-4}$	34
GMV	4	$2.4\ 10^{-3}$	$2.9\ 10^{-3}$	17
	8	$3.6\ 10^{-3}$	$3,2\ 10^{-3}$	11
	16	$4.2\ 10^{-3}$	$4.6\ 10^{-3}$	8
GMM	4	0.50	0.44	12
	8	1.80	2.03	11
	16	2.6.	3.01	13
PSRS	4	$2.5\ 10^{-3}$	$2.4\ 10^{-3}$	4
	8	$5.2\ 10^{-3}$	$4.8\ 10^{-3}$	8
	16	$6.1\ 10^{-3}$	$5.5\ 10^{-3}$	10

4.2. Performance analysis

Due to the space limitation, we only discuss the performances for two benchmarks: 1- The *Inprod* benchmark because it is the simplest one and has the same behaviour as GMM and GMV. 2- The *PSRS* benchmark as it has an interesting ratio computation/ communication and uses recursive calls for the hybrid version.

For the *inner product* benchmark with a small data size, the performance decreases when the number of processors increases. The communication time is larger than the computation time due to the synchronizations. Our tool determines the best configuration, which only uses OpenMP (1 node, 0 MPI, 2 OMP) as presented in Figure 5 with a predicted time of 6.5 10^-5 sec and the measured time is $7,1\ 10^-5$ sec. In all figures the $Hyb-OMP_i$ represent the hybrid codes with i threads per MPI process. With a large data size, as the computation time

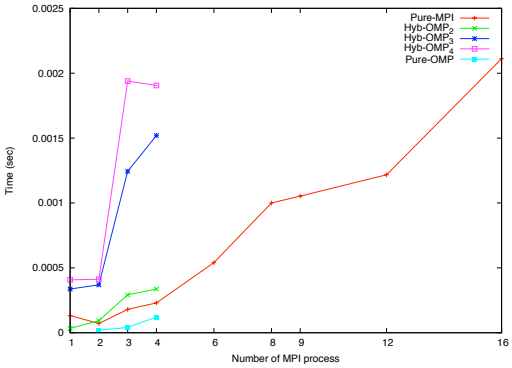

Figure 5. Execution time for all possible configurations for the inner product program with a small data size (64 k).

is bigger than the communication time and the synchronization is better with a hybrid model, the tool chooses the hybrid solution (4 nodes, 1 MPI, 4 OMP). Figure 6 shows that this configuration is the best one. Table 2 shows the times.

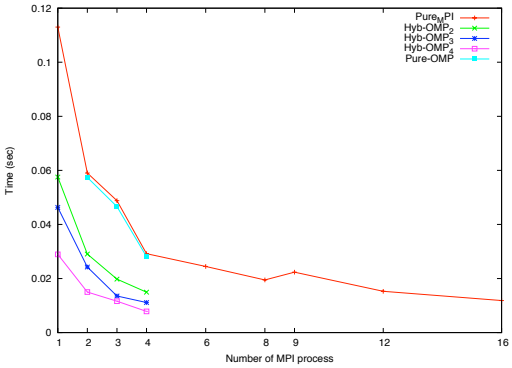

Figure 6. Execution time for all possible configurations for the inner product program with a large data size (128 M).

For the *PSRS* benchmark the situation is quite different. The BSP algorithm has 4 steps and within 3 of them there is a "*recursive*" call to a *Sort* function. As discussed before, each call to *sort* is replaced by the entire BSP program. This increases the computation time. The advantage on the communication time of the hybrid program does not compensate the increase of the computation time.

As for the inner product, with a small data size, the best configuration is Pure OpenMP program with 4 threads (figure 7). However, for a large data size, the ratio $\frac{parallel\ computation\ time}{parallel\ communication\ time}$ decreases with the number of processors. The fastest configuration uses all the processors in a pure MPI mode as shown in figure 8. The tool predicted times, the measured times and the error for these configurartions are depicted in table 2. The error is less than 16%.

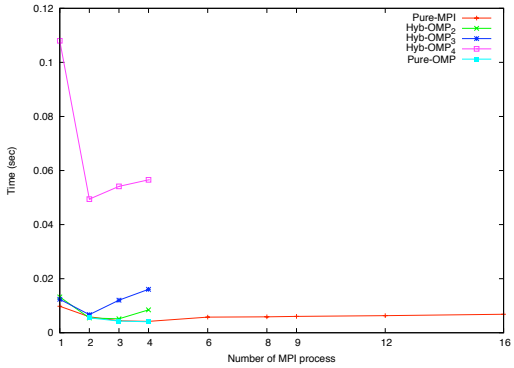

Figure 7. Execution time for all possible configurations for the PSRS program with a small data size (81920 elements).

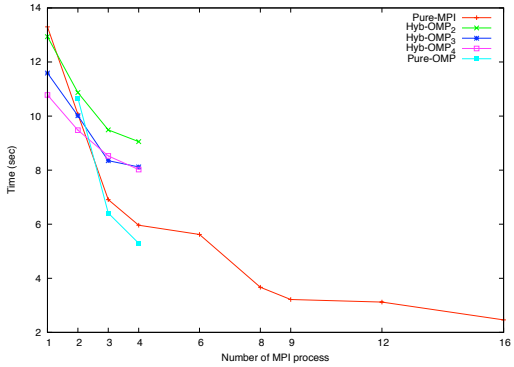

Figure 8. Execution time for all possible configurations for the PSRS program with a large data size (8192×10^4 elements).

Table 2. Comparison of measured and predicted times for the bests configurations

$Bench/size$	$Estim_{Time}$	$Meas_{Time}$	$Error\%$
$InProd/small$	$6.5\ 10^{-5}$	$7.5\ 10^{-5}$	8
$InProd/big$	$1.0\ 10^{-2}$	$1.3\ 10^{-2}$	16
$PSRS/small$	$4\ 10^{-3}$	$3.6\ 10^{-3}$	10
$PSRS/big$	2.46	2.66	7.5

5. CONCLUSION

In this paper, we have presented a framework to model, evaluate and generate parallel OpenMP, MPI and hybrid MPI+OpenMP programs. Our approach use a combination of a static analysis carried out by the *Analyzer* and feedback from the runtime by the *System profile* for communication and overhead measurements. The static analysis performed by *Analyzer* aims to estimate the computation time while the runtime benchmarks used by Sphinx and $prob-benchmark$ are used to retrieve the machine and network information.

Our future works involve: 1) The implementation of

BSP++ on a larger set of multi-core and many-core architectures including the Cell processor and GPGPUs. 2) In a similar way, the framework will be extended to generate hierarchical heterogeneous MPI/OpenMP/GPU programs and/or MPI/cells programs.

REFERENCES

[1] Clang. *http://clang.llvm.org/*.

[2] Sphinx. *http://www.llnl.gov/casc/sphinx/sphinx.html.*.

[3] ADHIANTO, L., AND CHAPMAN, B. Performance modeling of communication and computation in hybrid mpi and openmp application. *Parallel and Distributed systems ICPADS* (2006), 6.

[4] ALEXANDROV, A., IONESCU, M. F., SCHASER, K. E., AND SCHEIMAN, C. Loggp: incorporationg long messages into the logp model: one step closer towards a realistic model for parallel computation. *ACM symposium on Parallel Algorithms and Architecture (SPAA)* (1995), 95–105.

[5] BENKNER, S., AND SIPKOVA, V. Exploiting distributed-memory and shared-memory paralelism on clusters of smps with data parallel programs. *International Journal of Parallel programming* (2003), 3–19.

[6] BISSELING, R. H., AND MCCOLL, W. F. Scientific Computing on Bulk Synchronous Parallel Architectures. *Proc. 13th IFIP World Computer Congress* (1994), 31.

[7] BUSH, I., NOBLE, C. J., AND ALLAN, R. J. Mixed openmp and mpi for parallel fortran applications. *European Workshop on OpenMP (EWOMP)* (2000). Edinburgh, UK.

[8] CAPPELLO, F., AND ETIEMBLE, D. MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks. *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing* (2000), 12.

[9] CASCAVAL, C., DEROSE, L., PADUA, D., AND REED, A. Compile-time based performance prediction. *International Workshop on Languages and Compilers for Parallel Computing* (1999).

[10] CHOW, E., AND HYSOM. Assessing performance of hybrid mpi/openmp programs on smp cluster. *technical report UCRL-JC* (2001).

[11] CULLER, D., KARP, R., PATTERSON, D., SAHAY, A., SCHAUSER, E., SANTOS, E., AND TICKEN, V. Logp: towards a realistic model of paralel computation. *ACM SIGPLAN symposium on Principles and Paractice of parallel programming (PPOPP)* (1993), 12.

[12] GESBERT, L., AND GAVA, F. New Syntax of a High-level BSP Language with Application to Parallel Pattern-matching and Exception Handling. Tech. Rep. TR-LACL-2009-06, LACL (Laboratory of Algorithms, Complexity and Logic), University of Paris-Est, 2009.

[13] GIRAUD, L. Combining Shared and Distributed Memory Programming Models on Clusters of Symmetric Multiprocessors: Some Basic Promising Experiments. *International Journal of High Performance Computing Applications 16* (2002), 425–430.

[14] HAMIDOUCHE, K., FALCOU, J., AND ETIEMBLE, D. Hybrid bulk synchronous paralelism library for clustered smp architectures. *ACM Intenational workshop on High-Level parallel programming (HLPP)* (2010), 8.

[15] KIELMANN, T., BAL, E., AND VERSTOEP, K. Fast measurement of logp parameters for message passing platforms. *IPDPS workshop* (2000), 1176–1183.

[16] KRAWEZIK, G., AND CAPPELLO, F. Performence Comparaison of MPI and three OpenMP Programming Styles on Shared Memory Multiprocessors. *ACM Symposium on Parallel Algorithms* (2003), 118–127.

[17] LIAO, C., AND CHAPMAN, B. Invited paper: A compile-time cost model for openmp. *Parallel and distributed Processing symposium* (2007).

[18] MAJUMDAR, A. parallel performance study of monte carlo photon transport code on shared-, distributed- and distributed-shared-memory architectures. *IPDPS* (2000), 93.

[19] REYES, R., DORTA, A., ALMEIDA, F., AND SANDE, F. Automatic hybrid mpi+openmp code generation with llc. *European PVM/MPI* (2009), 185–195.

[20] SNAVELY, A., CARRINGTON, L., WOLTER, N., LABARTAAND, B., AND PURKAYASTHA, A. A framework for performance modeling and prediction. *ACM/IEEE conference on Supercomputing* (2002).

[21] VALIANT, L. G. A Bridging Model for Parallel Computation. *Commun. ACM 33*, 8 (1990), 103–111.

[22] WALLCRAFT, A. J. SPMD OpenMP versus MPI for Ocean Models. *concurrency: Practice and Experience 12* (2000), 1155–1164.

[23] WOLF, F., AND MOHR, B. Automatic performance analysis of hybrid mpi/openmp applications. *Euromicro conference* (2003).