Parallel Programming Models: A Survey

Rim Ketata, Lobna Kriaa, Leila Azzouz Saidane
CRISTAL Laboratory RAMSIS Pole - National School for Computer Sciences (ENSI)
Manouba, Tunisia

Abstract—**Parallel programming and the design of efficient parallel programs is a development area of growing importance. Parallel programming models are almost used to integrate parallel software concepts into a sequential code. These models represent an abstraction of the hardware capabilities to the programmer. In fact, a model is a bridge between the application to be parallelized and the machine organization. Up to now, a variety of programming models have been developed, each having its own approach. This paper enumerates various existing parallel programming models in the literature. The purpose is to perform a comparative evaluation of the mostly used ones, namely MapReduce, Cilk, Cilk++, OpenMP and MPI, within some extracted features.**

Keywords—Parallel programming models; parallel computing; MapReduce; Cilk/Cilk++; OpenMP; MPI.

I. INTRODUCTION

Over the last decades, the need to solve large problems in different fields such as sciences, engineering or business has been increasingly developed. Moreover, these problems may process large amount of data, which can reach the order of terabytes or even petabytes. Other applications are very highly compute-intensive, and involve applying a multitude of complex treatments. These kinds of problems require powerful platforms that enable them to accomplish their tasks in a reasonable amount of time. On the other side, Moore's law – the processor speeds will double every two years – has reached its physical limits and therefore, became no more applicable. In fact, increasing the processors frequencies beyond a particular threshold would result in energy dissipation problems. This had lead researchers to turn to parallel architecture designs in order to satisfy advanced programs requirements. However, all parallel characteristics of these new architectures are misused due to the sequential programming style of our algorithms. In this context, parallel computing has emerged as the alternative solution to exploit the maximum use of the underlying parallel architecture, thus enabling to run such applications efficiently, reliably and quickly. "It strikes me that in terms of future development, the magnitude of the change that software developers are going to experience will be substantial. A decade from now, we'll be looking back and thinking how much differently we approach writing program code. Parallelism, for everyone, is going to be ubiquitous." confirmed James Reinders, director of marketing and business in Intel Software Development Products Division.

Parallel programming models (PPM) aim to provide a mechanism with which the programmer can specify parallel programs. In the recent years, a variety of parallel programming models have been founded providing several development tools in order to achieve the best performance possible of parallel applications. Each model has its proper strategy to treat the problem, as a consequence, has its advantages as well as limitations.

In the following, we present the most significant parallel programming models existing in the literature, including MapReduce, Cilk, Cilk++, OpenMP and MPI. Each model is explained by an example that illustrates using of its paradigms to implement parallel algorithms. Section 2 gives a brief overview of MapReduce some basic concepts. Section 3 presents the fundamental idea upon Cilk and Cilk++. Section 4 deals with the main principles of OpenMP. Section 5 describes the MPI programming model. Section 6 discusses some criteria by which the mentioned PPMs can differ aiming at their classification.

II. MAPREDUCE

MapReduce was originally introduced by Google to run on a cluster of machines. It targets data intensive application. In the recent years, it has emerged as one of the most powerful and widely used parallel computing platforms for processing big data. Over 70 companies including Facebook, Yahoo, Adobe and IBM have adopted Hadoop, an open-source implementation of MapReduce [5].

A. Description

As the name implies, MapReduce is based on two main phases: the *map* phase, and the *reduce* one. In the map phase, data is first split into chunks with fixed data size (typically 64 megabytes), then passed to the "map" function as sets of <key,value>, denoted $<k_i,\ v_i>$. The "map" function, written by the user, takes a pair $<k_1,\ v_1>$, performs the specified treatments to generate sets of intermediate pairs list$<k_2,\ v_2>$. Note that this operation is done in parallel by many workstations forming the cluster. Then, the run-time system groups together, all intermediate values, from the output of the map function, sharing the same key as input to the "reduce" function. The later, also written by the user, takes a key and a list of its corresponding value $<k_2$, list$<v_2>>$, applies the specified treatments to produce a smaller set of values (typically one value) [1].

$$\text{Map} <k_1, v_1> \rightarrow \text{list} <k_2, v_2>$$
$$\text{Reduce} <k_2, \text{list} <v_2>> \rightarrow \text{list} <v_2>$$

It is important to mention that the input keys and values types are usually different from those of the output, unlike the intermediate keys and values which are from the same type of the output ones.

For a better understanding of MapReduce, the following example concretizes the theoretical aspects described above. It illustrates an application that counts the number of occurrences of each word in a set of text documents.

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
    result += ParseInt(v);
    Emit(AsString(result));
```

Fig. 1. WordCount example implemented in MapReduce [1]

The map function sends each word as a key followed by the value 1, whereas the reduce function sums together all the values corresponding to the same word.

B. Google implementation

When the user calls the MapReduce program, the later starts many copies of the program consisting in a master, and workers split into mappers and reducers. It's the master job to assign a worker either a map or reduce task. Once assigned, mappers begin execution, store intermediate values into local disks and notify the master about their locations. This last invoke the reducers which perform remote reads, accomplish their execution then notify the master. Note that each single reduce function may be assigned many different keys, but produces a single output file stored in the global file system.

In addition to the basic functionalities offered by MapReduce, it provides several extensions that contribute to enhance its performances. One of the most important ones is the "combiner" function. Once the map function has accomplished its task, the "combiner" function, also written by the user, performs a partial merging of data. Typically, it carries out the same work as the reduce function, but locally on the mapper's machine. This is useful to save the network bandwidth needed for data transfer.

Many applications are compliant to this programming model and can be expressed using MapReduce, such as distributed Grep, Count of URL Access frequency, Reverse Web-link Graph, inverted index, distributed sort, etc. [1, 3]

At first sight, MapReduce seems to cover only applications that are computationally straightforward. But for some cases, the overall distributed problem can be split into not just single map and reduce phase, but it can handle a sequence of map-reduce phases, so that the output of a single map-reduce step is the input of the next map-reduce algorithm. Many real-world applications are adopting such a mechanism, namely the Google indexing system.

III. CILK / CILK++

A. Cilk

Cilk is a *shared memory* task based language that facilitates the development of parallel applications. Initially founded by the MIT (Massachusetts Institute of Technology) laboratory for computer science, it was among the pioneering models that invoke multithreaded programming style. Cilk provides a set of extensions to the C language that enables the developer to create, synchronize and schedule threads. It presents an efficient tool to write dynamic, asynchronous, tree-like, MIMD computations.

As shown in "Fig. 2," a Cilk computation can be simulated as a directed acyclic graph (DAG) where each vertice corresponds to a thread. Sequences of threads connected horizontally constitute procedures forming the Cilk program. Once it has been invoked, a Cilk thread can run to completion without any suspending. As long as a thread proceeds running, it may alternatively spawn a child, connected to its parent thread by a downward edge. This is similar to a subroutine call, with the difference that a spawn child will not return any value to its parent. Rather, the parent spawns in parallel a successor that receives the return value of the child. In the DAG, the successor is linked horizontally to the creating thread. Note that the successor cannot begin execution unless the missing parameters are received. This creates data dependencies between threads, represented by an upward edge in the DAG.

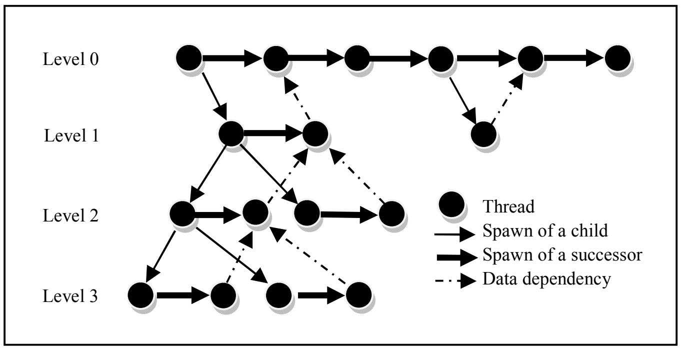

Fig. 2. Directed acyclic graph of a Cilk computation

1) Implementation

Cilk represents a thread as a "closure" that handles a data structure involving: a pointer to the C function for the thread, a set of its arguments as well as a "join counter" dealing with the number of missing arguments. A closure may have two possible states: it is "ready" if the number of missing arguments is equal to zero, "waiting" otherwise. Another data type provided by the Cilk language is the continuation, defined by the keyword "cont". The later represents a global reference to an empty argument slot of a closure. Having the same type of the argument it references, this structure is used to communicate data between threads.

During execution, a thread can spawn a child using the function "spawn". It can possibly spawn a successor within the function "spawn_next". Usually, a parent procedure containing a subroutine call is coded as two threads. The first spawns a child procedure, passing it a continuation pointing to the successor thread's closure. Missing arguments are specified inside the spawning function by the name of the argument preceded by a question mark. The child thread sends data value to the waiting successor by means of the function "send_argument". As a consequence, the corresponding counter of the successor is decremented. This procedure is called "explicit continuation passing".

Fig. 3 shows a Cilk program that computes the maximum value of elements in a table.

```
thread tab_max(cont int res, int T[], int begin, int end) {
 if (begin != end) {
  int middle = (begin + end) div 2;
  cont max_left, max_right;
  spawn_next t_max(res, ?max_left, ?max_right);
  spawn tab_max(max_left, T, begin, middle);
  spawn tab_max(max_right, T, middle + 1, end);
 }
 else
 {
  send_argument(T[begin]);
 }
}

Thread t_max (cont int res, int max_left, int max_right)
{
 send_argument(res, max(max_right, max_left));
}
```

Fig. 3. Example of a Cilk program

2) Work-stealing scheduler

Cilks's runtime system uses a work-stealing scheduler, where each processor running out of work (thief) steals work from another processor (victim).

Each processor has a local queue that contains ready closures. For a ready closure is assigned a level corresponding to the number of spawns (within the level axis in fig. 2) beginning from the root (level 0). A ready queue is an array whose L'th element is a linked list of ready closures having level L and assigned to the processor. A thief chooses a victim processor at random. Whenever chosen, it selects a closure from the list with the lowest level in the processor's local queue. This mechanism allows the thief to take a large amount of work from the victim as much as possible, since the stolen thread may spawn children while running, thus reducing the overall time required for the program execution.

As a cilk program begins execution, it places the root thread in the ready queue of a processor. Note that all ready queues are initially empty. While running, a thread with the level L may spawn a child or a successor. In this case, the scheduler allocates a closure for the new thread, fill in arguments as well as the join counter to the number of the missing arguments. Alternatively, it may initialize continuations to point to the missing arguments. Otherwise it attributes the level L+1 to the child thread (resp. L to the successor) before posting them to the processor's ready queue.

When a processor terminates a thread execution, it checks the existence of other threads in its ready queue, from which it selects the one having the highest level. Otherwise it performs work stealing.

3) Performance

Cilk guarantees to the user two major performance quantities to characterize the performance of applications. The first one, called the "work", corresponds to the minimum amount of time needed for serial execution of the multithreaded application (i.e. on one processor). The second measure, called "critical path", is the minimum execution time of the same program on an infinite number of processors [23], which corresponds to the longest path execution time among any path of the DAG. In fact, these quantities can be used to predict the runtime of a Cilk program. Denoting T_1 as the work, T_∞ as the critical path, Cilk guarantees that the execution time of the program on P processors is very near to the sum of these two measures [23].

B. Cilk++

Similarly like Cilk, Cilk++ is an extension to the C++ programming language that offers a reliable way for multithreaded parallel programming. Intuitively, the developer does not need to restructure applications significantly in order to add parallelism.

Cilk++ was originally built by Cilk Arts, after it was licenced the Cilk technology by the MIT laboratory. Today, both open sources as well as commercial versions are available, after Intel has acquired Cilk Arts.

1) Description

Cilk++ programming language can be merely summarized in three key words: "cilk_spawn", "cilk_sync" and "cilk_for". The "cilk_spawn" statement precedes the invocation of a function, thus allowing the creation of a spawn child. The "cilk_sync" statement acts like a barrier, where all children must achieve the same point and complete execution before the caller proceeds. Unlike other programming models, this barrier is local to its calling function, meaning that all threads created by this function must reach this execution point without affecting other threads spawn by other functions. Note that any function embodies an implicit "cilk_sync" at its end, ensuring all its spawned children terminate before it does. The "cilk_for" statement represents a major improvement of Cilk++ upon Cilk. It enables automatic parallelization of loop's iterations. Note that a "cilk_for" can be expressed as a for loop using cilk_spawn and cilk_sync in MIT Cilk system to create parallelism over iterations. The Cilk++ runtime system provides several other enhancements such as mutual exclusion locks, support for exceptions etc. The example in fig. 4 can help understand the basic Cilk++ concepts. It contains the same function as in the previous paragraph, but written with Cilk++.

```
int tab_max(int T[], int begin, int end) {
 if (begin != end) {
  int max_left, max_right;
  int middle = (begin + end) div 2;
  max_left = cilk_spawn tab_max(T, begin, middle);
  max_right = tab_max(T, middle + 1, end);
  cilk_sync
  return max(max_left, max_right);
 }
 else
 {
  return(T[begin]);
 }
```

Fig. 4. Example of a Cilk++ program

In this example, "cilk_sync" acts like a local barrier where the calling function must wait the completion of its children, corresponding to the calculation of the maximum of its two split sub-tables, before it returns. This is mandatory to guarantee the consistency of the final result. This synchronization is achieved by means of the continuation type in the previous example written with Cilk.

As Cilk++ is an evolution of the MIT Cilk model, it shares the same principles on which Cilk is based like the work stealing scheduler and the directed acyclic graph. In fact, a cilk_spawn invocation of a function creates two edges: ones goes from the instruction preceding the cilk_spawn, whereas the other goes to the first instruction of the spawned function. The cilk_sync creates edges from the final instruction of each spawned function to the instruction following the cilk_sync [26].

2) Hyperobjects

Cilk++ introduces the concept of hyperobjects to handle data race bugs in code with non local variables. A data race takes place when two strands (sequential execution of instructions) have access to the same shared location, outside the use of any lock, with at least one strand modifies the value of the location. Although using nonlocal variables simplifies the coding task for the programmer, their management remains a problematic issue in the context of parallel computing. Applying traditional methods like the mutual exclusion lock is unfortunately unpractical and inefficient since it may cause bottleneck during execution. Rather, Cilk++ provides additional entities called “hyperobjects” that allows different strands to have their local view of the same nonlocal variable. During the strand’s execution, these views are private so that the threads can modify their states without any synchronization. When different strands join, they coordinate to update the value of the shared variable. Several hyperobjects are supplied by the Cilk++ runtime system, namely the “reducers”. When the branches synchronize, a reducer performs a combining operation to local views of each thread and the result is assigned to the shared variable. Unlike other programming models, the reduction operation can be fulfilled progressively as long as some strands are done and does not require that the overall threads to be completed. Besides, the reduction function has only to be associative, commutativity is an option. Another type of hyperobjects provided by the Cilk++ runtime system is the “splitter”. This last is useful for some cases where a global variable exhibit no change from immediately before a cilk_spawn to immediately after the cilk_spawn when executed serially, either because the subroutine does not change the variable’s value, or it turns it back after modification before the function returns. An example of such variable is the local depth of a node inside a binary tree. In this particular case this variable can be declared as a splitter so as to attribute different views to the various nodes during tree traversal [27].

IV. OPENMP

OpenMP is considered as one of the most popular *shared memory* parallel programming models in use today. Several vendors are creating products as compilers, development tools and performance analysis tools for OpenMP, including Intel, IBM, HP, Fujutsu, AMD, ARM, etc. OpenMP has been well used to standardize SMP machine programming over the last fifteen years.

OpenMP Application Programming Interface is a portable standard which offers a simple and straightforward programming environment to write parallel multithreaded applications. In fact, a developer can easily transform a sequential code into a parallel one. This is simply done by inserting a set of directives into the sequential code. OpenMP is not a language by itself, but an extension to existing languages like C, C++ and Fortran.

It is important to say that it is not mandatory to parallelize the entire application. Instead, just putting directives to loops or statements in the programs is enough to parallelize some regions of the code, thus offering an incremental approach of parallelization. In practice, OpenMP is principally used to parallelize loops which are too much time-consuming.

A. The OpenMP Application Programming Interface

OpenMP provides several directives that can be classed into three main types dealing with parallelism / work sharing, data environment and synchronization.

- Parallel directives: OpenMP uses the fork-join model of parallel execution. An OpenMP program starts execution as a single thread, called the master thread. The “parallel” construct is the fundamental directive to express parallelism. When the master reaches this point of execution, it creates a number of child threads with the fork statement. The child threads along with the master work in parallel and execute a given part of the program. By leaving the parallel region all threads synchronize and join the master. The master then proceeds execution of the serial code, whereas the remaining threads terminate.
  Work-sharing directives, like the “for” directive, split the overall set of iterations into threads, each of which processes a subpart of the computation. “Section” directive attributes a different structured block to a different thread.
- Data environment directives define the visibility inside a parallel region of variables declared outside this block. They specify whether these variables are “shared” by all threads, or “private”. In the last case, each thread creates a local copy of the variable. Unless explicitly specified, variables default to shared since OpenMP works on shared memory architecture. In the opposite side, variables of a subroutine stack called inside a parallel block are private.
- Synchronisation: several synchronization directives are provided by OpenMP. The “critical” directive allows only to one thread to enter a region at the same time. A particular case of critical sections is the “atomic” directive. It is used only for assignment to ensure that the specific storage location is accessed atomically. The “barrier” directive states that all threads must wait each others to reach this point of execution before proceeding. By default, an implicit barrier resides at the end of parallel blocks like the “for” loop, the “parallel” bloc, the “section” bloc. The “nowait” clause removes this barrier.

Applications written using OpenMP directives can be compiled by OpenMP compilers as well as non-OpenMP compilers. Besides, OpenMP give the possibility to build a program without directives interpretation.

B. Example

Fig. 5 shows a simple OpenMP pseudo-code where three threads execute distinct functions in parallel. Each thread executes a different code block delimited by the directive section. An implicit wait is existing at the end of the section region. After they join (when leaving the parallel section region) the master runs a function that depends on the result of parallel threads execution, and uses the last result to run another function.

```
main(int argc, char **argv) {
  int a, b, c s;
  #pragma omp parallel sections
  {
  #pragma omp section /* Optional */
     a = alice();
  #pragma omp section
     b = bob();
  #pragma omp section
     c = cy();
  }
  s = boss(a, b);
  printf("%6.2f\n", bigboss(s,c));
}
```

Fig. 5. Example of a program implemented in OpenMP

V. MPI

Message Passing Interface (MPI) is at present the most widely adapted framework for programming parallel applications for *distributed memory* and clustered parallel systems [14]. Over 40 organizations, between vendors, developers, researchers and users are members of the MPI forum.

MPI is a standard for message passing models having a variety of implementations. It defines the manner how its features should behave among different implementations. Moreover, MPI is a portable API which supports almost every distributed memory architecture, meaning that there is no need to modify your code when you port your application to a different platform that is compliant with the MPI standard. MPICH and Open MPI are examples of implementation of MPI. Besides, MPI handles both SPMD and MPMD execution modes of parallel units.

In this message passing model, a parallel program is a combination of processes performing cooperative operations, working on different address spaces (In contrast to threads that share a common address space, thus are the most suitable for shared memory architecture). In fact, processes send data to one another as messages that may have tags useful to sort them.

A. The MPI Application Programming Interface

An MPI program is organized as follows: An alternative serial code may be existing at the beginning and/or the end of the program. Parallel region of the code is managed by several routines. MPI_Init (resp. MPI_finalize) is used to initialize the parallel execution environment (resp. to terminate the parallel execution environment). It is important to mention that these functions can be called once – and only once – in an MPI program.

MPI uses groups to define a collection of processes that communicate with each others. A "communicator" per group is mandatory to enable processes communication within the same group. By default, "MPI_COMM_WORLD" is the predefined communicator that contains all processes of the program. When the process initializes, a corresponding rank for every communicator is attributed by the system. In other words, a process may have several ranks (or identifiers), but once per communicator. Note that data exchange is explicitly specified by the programmer, and ranks are useful for him to define source and destination processes whenever he needs to send or receive a message.

Communication operations within a group involve point-to-point as well as collective communication routines. In point-to-point operations messages are passed between only two cooperative processes: the first performs a send and the other carry out the matching receive operation. Point-to-point routines can be classified into several types: blocking / non-blocking send, blocking / non-blocking receive, combined send / receive, synchronous send, buffered send, etc. Take into consideration that the last type is useful for asynchronous communication. The system buffer space is completely managed by the MPI library so that it is fully opaque to the programmer. Collective communication routines involve all processes scoped to a group. They can be classified into three types: synchronization, data movement and collective computation. In synchronization operations, a process must wait until other processes belonging to the same group reach a synchronization point. Data movement operations involve message broadcast, scatter / gather, etc. In collective computation, a member of the group collects data from the other members and performs an operation on that data (such as min, max, add, multiply, etc.). It is the programmer's responsibility to check that all processes within a group participate in the collective operation.

In addition to the intracommunication operations previously described, MPI offers the possibility for different applications to exchange information via intercommunicators, thereby allowing to processes belonging to different groups to intercommunicate.

B. Example

The example shown in fig. 6 illustrates a simple MPI program where two processes exchange messages, perform some treatments and print out their rank as well as the message received. The rank indicates the identity of the running process.

The MPI_Comm_size routine returns the size of the group associated with a communicator, whereas the MPI_Comm_rank determines the rank of the calling process in the communicator. The MPI_Send (resp. MPI_Receive) performs a blocking send operation (resp. a blocking receive operation).

In this example, two processes having ranks 0 and 1 compute the sum of elements of an array table. The first process (with the rank 0) reads the number of elements as well as their values from the standard input. Next, the work is split between both processes: the one ranked 0 sends the half portion of the elements introduced by the user to the process with rank 1. Working in parallel, each process computes a partial sum of the portion assigned to him. Once completed, the process ranked 1 sends the result back to the first process which computes the total sum and prints the final value to the standard output. Information exchange is fulfilled using blocking send/receive. In the example, partial_sum refer to two different variables located in two different address spaces (once per process).

```
#include "mpi.h"
#include <stdio.h>
#define MAX_SIZE 10000

main(int argc, char **argv) {
  int numtasks, rank, dest, source, rc, tag=1;
  int i, sum, partial_sum, num_rows, num_rows_to_send,
num_rows_to_receive;
  int array[MAX_SIZE], partial_array[MAX_SIZE];
  MPI_Status Stat;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
   dest = 1;
   source = 1;

   /* reads the input elements of the table */
   printf("please enter the number of elements to sum: ");
   scanf("%i", &num_rows);
   printf("please enter the values of elements to sum: ");
   for(i = 0; i < num_rows; i++) {
     scanf (“%i”, &array[i]);
   }

   /* send the half number of the table elements to the process with
rank 1 */
   num_rows_to_send = num_rows div 2;
   rc = MPI_Send(&num_rows_to_send, 1, MPI_INT, dest, tag,
MPI_COMM_WORLD);
   rc = MPI_Send(&array[0], num_rows_to_send, MPI_INT, dest, tag,
MPI_COMM_WORLD);

   /* computes the sum of the remaining elements in parallel with the
process ranked 1 */
   sum = 0;
   for(i = num_rows_to_send; i < num_rows; i++) {
     sum += array[i];
   }
   printf("Task %d: The partial sum performed by task %d  is %d \n",
rank, rank, sum);


   /* receives the partial sum of elements computed by the process
with rank 1 */
   rc = MPI_Recv(&partial_sum, 1, MPI_INT, source, tag,
MPI_COMM_WORLD,   &Stat);
   sum  += partial_sum;
   printf("Task %d: The partial sum performed by task %d is %d \n",
rank, Stat.MPI_SOURCE, partial_sum);

   /* print the final result */
   printf("Task %d: The result of treatment is %d \n", rank, sum);
   }

  else if (rank == 1) {
   dest = 0;
   source = 0;

   /* receive the table elements to sum */
   rc = MPI_Recv(&num_rows_to_receive, 1, MPI_INT, source, tag,
MPI_COMM_WORLD,   &Stat);
   rc = MPI_Recv(&partial_array, num_rows_to_receive, MPI_INT,
source, tag, MPI_COMM_WORLD,   &Stat);

   printf("Task %d: Received %d elements to sum from task %d \n",
    rank, num_rows_to_receive, Stat.MPI_SOURCE);

   /* computes the partial sum of the elements received */
   partial_sum = 0;
   for(i = 0; i < num_rows_to_receive; i++) {
     partial_sum += partial_array[i];
   }

    /* sends the partial result back to the process with rank 0 */
   rc = MPI_Send(&partial_sum, 1, MPI_INT, dest, tag,
MPI_COMM_WORLD);
   }


  MPI_Finalize();
}
```

Fig. 6. Example of an MPI program consisting of two parallel processes computing the sum of an array table elements

VI. COMPARING THE PROGRAMMING MODELS

Since its establishment, most of parallel computing is still being done through multithreading and message passing. In fact, all of the previously described parallel programming models have proven a great success from their creation up to now, each one having its specificity. Depending on the application needs and the target architecture, some particular PPM would be the most suitable than others.

While Cilk, Cilk++ and OpenMP work on platforms with shared memory architecture, distributed memory machines are strongly connected to message passing models like MPI. Although MapReduce was originally developed on distributed memory, this model has been ported to shared memory systems [3].

There are several other criteria by which the parallel programming models can differ. Table 1 summarizes some of these criteria.

Scheduling consists in assigning tasks to processes or threads, thus fixing the order of their execution. This assignment can be done either statically at program start, or dynamically during execution. It is different from mapping which refer to the attribution of processes or threads to physical units (processors, cores).

Determinism is the use of automatic synchronization tools provided by some parallel programming models without any programmer intervention. The explicit way leaved to the developer to synchronize threads in order to ensure the correct execution of the program is named indeterminism [19].

Both MapReduce and OpenMP provide a simple way and easy to use framework to write parallel applications, unlike MPI where the programmer is responsible for determining all details of parallelism, including communication and synchronization between processes. For MapReduce, all these low-level details are hidden so that the programmer just needs to concentrate on the sequential code. This implicit representation of parallelism is what makes it an easy and attractive tool. But it handles applications that can only be expressed in particular way (i.e. following MapReduce pattern). Despite the fact that parallel application development using Cilk/Cilk++ is not as straightforward as other programming models, it is suited for divide and conquer applications - namely the recursive ones - where problems can be divided into parallel independent tasks and the results can be

TABLE I. COMPARING THE PARALLEL PROGRAMMING MODELS

Criteria	Parallel Programming Models			
	MapReduce	***Cilk/Cilk++***	***OpenMP***	***MPI***
Organisation of the address space	Distributed / Shared	Shared	Shared	Distributed
Representation of parallelism	Implicit	Explicit	Explicit	Explicit
Data level parallelism	Yes	Yes	Yes	Yes
Task level parallelism	No	Yes	Yes	Yes
Data decomposition support	Yes	No	Yes	No
Incremental parallelism	No	Yes	Yes	No
Deterministic	Yes	No	No	No
Built-in load balancing	Yes	Yes	Yes	No
Supported languages	All	C/C++	Fortran, C/C++	Fortran, C/C++
Support for parallel programming	Library	Extension	Compiler directives	Library
Static / dynamic scheduling	Static	Dynamic	Static / Dynamic	Static / Dynamic
Static / dynamic mapping	Dynamic	Dynamic	Static / Dynamic	Static / Dynamic
Complexity	Simple	Complex	Simple	Complex
Level of Abstraction	High	Middle	Middle	Low

combined afterward. Although OpenMP requires an explicit parallelism, it resides as a simple model allowing an incremental parallelization of an existing code. This model is useful for programmers who need to transform quickly a sequential code into a parallel one.

VII. CONCLUSION

In this paper, we gave a brief overview of several leading parallel programming models existing in the literature. We introduced the basic concepts behind these models, followed by a comparison of some extracted features for each one. MapReduce is well known for its simplicity but efficiency to solve big data applications. Cilk/Cilk++ is a shared memory programming model well suited for problems based on divide and conquer strategy. Even though OpenMP is also designed for shared memory, it remains the standard that provides an incremental approach to the parallelization of a sequential code. On the other hand, MPI is the first appeared standard application programming interface for distributed memory architecture.

These programming models are not mutually exclusive and can be combined to yield systems that exploit most benefits and make maximum use of techniques they provide. In fact, as a response to the huge advance of technology in computer sciences, and in order to achieve the best possible performance, axes of research tend to overlap more than one parallel programming model. For instance, using MPI to coarsely distribute work among machines, and using OpenMP to parallelize at finer level on a single machine. But the search for better models remains always a research topic.

Future works would include a more detailed study that embodies a performance analysis and allows a comparison of these models regarding other runtime criteria like execution time, speedup, etc. In addition, we intend to answer the following questions: “Can these PPMs be related, associated, and/or specific to application domains (such as imaging, networks, etc.)? Can PPMs be adapted so as to take into account variabilities from one domain to another? And if “Yes”, how should it be done?

REFERENCES

[1] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” in Proceedings of Operating Systems Design and Implementation, OSDI., pp. 137-150, 2004.

[2] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” Communications of the ACM, vol. 51 N°1, pp. 107-113, January 2008.

[3] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, “Evaluating MapReduce for multi-core and multiprocessor systems,” in HPCA ’07: Proceedings of the 13th International Symposium on High-Performance Computer Architecture, pp. 13-24, February 2007.

[4] Maitrey, S., Jha, C.K, “Handling Big Data Efficiently by using MapReduce Technique IEEE International Conference on Computational Intelligence and Communication Technology, CICT., pp. 703 - 708, February 2015.

[5] H. Karloff, S. Suri, S. Vassilvitskii, “A model of computation for MapReduce,” Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms, SODA ’10, pp. 938 - 948, 2010.

[6] Guo Yucheng, “MapReduce model implementation on MPI platform,” 13th International Symposium on Distributed Computing and Applications to Business, Engineering and Science, 2014.

[7] Surbie Wattal, S.B.L. Tripathi, Narender Kumar, “To study and explain the different concepts of parallel processing by parallel computing,” IJCSMS International Journal of Computer Science and Management Studies, Vol. 12, Issue 02, pp. 371 - 376, April 2012.

[8] B. Chapman, G. Jost, and R. v. d. Pas, Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press, 2008.

[9] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon, Parallel programming in OpenMP. 2001 by Academic Press.

[10] Xinmin Tian, Milind Girkar, Sanjiv Shah, Douglas Armstrong, Ernesto Su, Paul Petersen, “Compiler and runtime support for running OpenMP programs on Pentium- and Itanium-architectures,” Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS’03).

[11] A. Qawasmeh, A.M. Malik, B.M. Chapman, “OpenMP Task Scheduling Analysis via OpenMP Runtime API and Tool Visualization,” IEEE 28th International Parallel & Distributed Processing Symposium Workshops, IPDPSW., pp. 1049 - 1058, 2014.

[12] Ying Peng, Fang Wang, “Cloud computing model based on MPI and OpenMP,” 2nd International Conference on Computer Engineering and Technology (Volume 7), pp. 85 - 87, 2010.

[13] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, MPI: The Complete Reference. second ed., vol. 1. MIT Press, 1998.

[14] J.L. Traff, W.D. Gropp, R. Thakur, “Self-consistent MPI performance guidelines,” IEEE Transactions on Parallel and Distributed Systems, Vol. 21, N°. 5, pp. 698 - 709, May 2010.

[15] Samuel K. Gutierrez, Nathan T. Hjelm, Manjunath Gorentla Venkata, Richard L. Graham, “Performance evaluation of Open MPI on Cray XE/XK systems,” IEEE 20th Annual Symposium on High-Performance Interconnects, pp. 40 - 47, 2012.

[16] Humaira Kamal, Alan Wagner, “Added concurrency to improve MPI performance on multicore,” 2012 41st International Conference on Parallel Processing.

[17] R. Wottrich, R. Azevedo, G. Araujo, “Cloud-based OpenMP parallelization using a MapReduce runtime,” IEEE 26th International Symposium on Computer Architecture and High Performance Computing, pp. 335 - 341, 2014.

[18] Hisham Mohamed, Stephane Marchand-Maillet, “Enhancing MapReduce using MPI and an optimized data exchange policy,” 41st International Conference on Parallel Processing Workshops, pp. 11-18, 2012.
[19] Evgenij Belikov, Pantazis Deligiannis, Prabhat Totoo, Malak Aljabri, and Hans-Wolfgang Loidl, A survey of high-level parallel programming models. Technical Report HW-MACS-TR-0103: Heriot-Watt University, Edinburgh, December 2013.
[20] J. Diaz, C. Munoz-Caro, A. Nino, “A survey of parallel programming models and tools in the multi and many-core era,” ,” IEEE Transactions on Parallel and Distributed Systems, Volume 23, Issue 8, pp. 1369-1386, Juin 2012.
[21] M. Frigo, C. E. Leiserson, and K. H. Randall, “The implementation of the Cilk-5 multithreaded language,” In Proceedings of the ACM SIGPLAN ’98 Conference on Programming Language Design and Implementation, PLDI., pp. 212-223, Montreal, Quebec, June 1998.
[22] L. Peng, M. Feng and C.K. Yuen, “Evaluation of the performance of multithreaded Cilk runtime system on SMP clusters,” 1st IEEE International Workshop on Cluster Computing, IWCC '99, pp. 43-51, December 1999.
[23] R.D. Blumofe, C.E. Leiserson, “Scheduling multithreaded computations by work stealing,” Proceedings of the 35th Annual Symposium on Foundations of Computer Science, FOCS., Santa Fe, New Mexico, pp. 356-368, November 1994.
[24] R. Basseda, R.A. Chowdhury, “A parallel bottom-up resolution algorithm using Cilk,” IEEE 25th International Conference on Tools with Artificial Intelligence, ICTAI., pp. 95-100, 2013.
[25] Intel Corporation, Intel Cilk++ SDK Programmer’s Guide. October 2009. Document Number: 322581-001US.
[26] C. E. Leiserson, “The Cilk++ concurrency platform,” in DAC ’09: Proceedings of the 46th Annual Design Automation Conference, pp. 522-527, 2009. ACM.
[27] M. Frigo, P. Halpern, C. E. Leiserson, S. Lewin-Berlin, “Reducers and other Cilk++ hyperobjects,” Proceedings of the 21st annual Symposium on Parallelism in Algorithms and Architectures, SPAA ’09, pp. 79-90, 2009. ACM.
[28] A.D. Robison, “Composable parallel patterns with intel Cilk plus,” Computing in Science & Engineering, Volume:15 , Issue: 2, pp. 66-71, 2013.
[29] S. Tham and J. Morris, “Cilk vs MPI: comparing two very different parallel programming styles,” Proceedings of the IEEE International Conference on Parallel Processing, ICPP’03, pp. 143-152, 2003.