AN INVESTIGATION OF MULTI-CORE (SINGLE-CHIP MULTIPROCESSOR) PROGRAMMING LANGUAGES

Pablo Ibáñez Verón

Supervisor: Farhang Nemati
Examiner: Thomas Nolte

November 18, 2010
ABSTRACT

The goal of this thesis is to overview the different multi-core programming models and technologies. Nowadays multi-core systems are more and more popular. Personal computers usually are based on multi-core technology since some years ago so it is needed some technologies for implementing it. In this thesis, we have studied several models and technologies of multi-core programming. We have also studied and presented a comparison of the models and technologies. Furthermore, we have shown the installation and configuration instructions for the two main technologies (OpenMP and MPI).
CONTENTS

CHAPTER 1: INTRODUCTION ..........................................................5

CHAPTER 2: OPENMP .................................................................7
  2.1 What is OpenMP? ....................................................................7
  2.2 Working sharing for loops in OpenMP .................................. 9
  2.3 Non-loop parallelism in OpenMP ......................................... 11
  2.4 Library functions and environment variables ....................... 15
  2.5 Conclusions of OpenMP ...................................................... 15

CHAPTER 3: MPI (MESSAGE PASSING INTERFACE) .......................17
  3.1 What is MPI? ......................................................................... 17
  3.2 Point-to-point communication ............................................. 20
  3.3 Collective operations ........................................................... 23
  3.4 Data types ........................................................................... 24
  3.5 Conclusions of MPI ............................................................. 26

CHAPTER 4: CODEPLAY SIEVE SYSTEM ......................................28
  4.1 What is Sieve ........................................................................ 28
  4.2 Implementation of Sieve ..................................................... 31
  4.3 Conclusions of Sieve ............................................................ 33

CHAPTER 5: CHAPEL ...................................................................34
  5.1 What is Chapel? ................................................................. 34
  5.2 Chapel principles ............................................................... 35
5.3 Conclusions of Chapel ................................................................. 37

CHAPTER 6: CILK++ ........................................................................... 38
6.1 What is Cilk++? ........................................................................... 38
6.2 Cilk++ working model ................................................................. 39
6.3 Cilk++ execution model ............................................................... 41
6.4 Conclusions of Cilk++ ................................................................. 42

CHAPTER 7: SKANDIUM ................................................................. 43
7.1 What is Skandium ....................................................................... 43
7.2 Skandium programming model .................................................... 43
7.3 Conclusions of Skandium ............................................................ 46

CHAPTER 8: A COMPARISON BETWEEN THE PROGRAMMING
TECHNOLOGIES ............................................................................. 47
8.1 Technology comparison table ....................................................... 47
8.2 Chapel limitations and Cilk++ vs. Skandium ............................... 48
8.2 The “Big Three”: OpenMP, MPI and Sieve ................................ 49

APPENDIX A: INSTALLATION AND CONFIGURATION OF OPENMP AND
MPI FOR C++ .................................................................................... 52
A.1 Installation and Configuration of OpenMP ................................. 52
A.2 Installation and Configuration of MPI ......................................... 56
A.3 Creation of a C++ profile .......................................................... 58

CONCLUSION ..................................................................................... 61

REFERENCES .................................................................................... 63
CHAPTER 1

INTRODUCTION

Applications have increased the capability of running multiple tasks simultaneously. Server applications today often consist of multiple threads or processes. In order to support this thread-level parallelism, several approaches, both in software and hardware, have been adopted.

One approach to address thread-level parallelism is to increase the number of physical processors in the computer in order to allow parallel execution; this means that multiple processes or threads run simultaneously on multiple processors. The disadvantage of this is that the overall system cost increases. However, Multi-core processors provide very high levels of processing power for very low levels of electrical power consumption.

Multi-core processors work with CMP (Chip Multiprocessing). Instead of reusing resources in a single-core processor, several cores inside a single chip are collocated. The cores are basically two individual processors on a single chip. The cores are provided with a set of architectural and execution resources. The processors could share a large on-chip cache but it depends on design. Moreover, the individual cores can be combined with SMT (Simultaneous Multi-Threading).

Owing to these multiprocessors there is a need of developing different parallel programming models for working with them. Parallel programming models are an abstraction above memory and hardware architectures.
The election of which model to use is usually determined by which technology is available and personal choice. It is not realistic to look for an ideal solution as there is no "best" model. However there are always better implementations than others in different situation.

Note that these technologies are not usually specific to a particular type of machine or memory architecture. In fact, any of these models should be able of being implemented on any hardware.

In this thesis we show how basically some of the most important models work and technologies however there are many more models and technologies which are not presented in this thesis as it is too difficult to analyze all of them within a master thesis.
CHAPTER 2

OPENMP [1], [2], [3]

2.1 What is OpenMP?

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in several platforms and architectures such as UNIX and Microsoft Windows platforms. It allows adding concurrence to C, C++ and FORTRAN by using fork-join. Fork-Join comes from UNIX systems in which a complex task is divided into K light threads which run separately and at the end they are joined to get only one result.

The basis of OpenMP is a set of compiler directives, library routines and environment variables that influence run-time behavior. It has been defined by big hardware and software development companies. The aim of its designers was to provide developers an easy way to create multi-thread applications without knowing when to create or destroy threads. It is achieved by using pragmas, clauses, and environment variables which advice the compiler to use threads. Usually the developer only needs to insert one single pragma before the loop to make parallelization run and adding a command-line option to the compiler instructs to make compiler to know that there are OpenMP pragmas. This is useful because the programmer can change a multithreaded code to single threaded by only omitting the command-line option.
CHAPTER 2: OPENMP

Because of this, OpenMP is a scalable and portable model for the programmers, which provides them a simple and flexible interface in order to develop parallel applications for different kind of platforms (from desktop to supercomputer). Thanks to OpenMP, a programmer can work with a classical supercomputer (like Cray) although it is possible to run OpenMP in every dual-core computer. The programmer develops in a sequential way so that it is easier and the parallel portion is executed by threads at the same time. Note that managing the threads is done by the compiler instead of the programmer. Moreover as constructs are clauses an OpenMP program could be compiled by non-OpenMP compilers.

In C++ the constructs are like this:

```cpp
#pragma omp construct [clause [clause]...]
```

Meanwhile in FORTRAN the constructs are written as follows:

```fortran
C$OMP construct [clause [clause]...]
```

From now on all examples we show will be C++ examples.

The following is an example of OpenMP implementation. Next piece of code shows how to make a simple execution to implement multi-thread.

```cpp
1. #pragma omp parallel for
2. for (i=97; i < numChars; i++)
3. {
4.    Upper List[i] = ASCII Table[i]-32;
5. }
```

First of all we can see the OpenMP pragma must be just before the loop. It uses work-sharing (distributing of work across threads) and the parallelization will be implemented in for construct. By this, each iteration is executed only once but, because there are several threads working at the same time, several iterations will be executed at the same time too. In this case, it converts lowercase characters from lowercase to uppercase characters. With this we see that programmer only needs one code line in order to implement OpenMP. However there are some restrictions when it works with loop iterations such us that variable must be signed integer or the loop must be a basic loop (exit is not allowed, and goto and break are restricted to work only inside the block).
2.2 Working sharing for loops in OpenMP

Although the restrictions mentioned before are fulfilled there still could be problems. One of the most common problems when it comes to developing an OpenMP program is data dependencies. It means that two or more iterations of a loop write in a shared memory at the same time. An example of this is as follows:

1. #pragma omp parallel for
2. for (i=2; i < 10; i++)
3. |
4. factorial[i] = i * factorial[i-1];
5. |

If we make compiler to execute this code it will do it without any warning sign but it will fail because some loop iteration depends on other iteration data. This is called race condition. In order to solve the problem the programmer should use a different algorithm which do not match race condition or rewrite the loop. However finding race conditions is not easy because in one single instance the variables can run in the right order but it could not always be the same so it should be tried in different machines to detect the race condition.

Another problem could happen when a programmer uses shared and private variables in a wrong way. In most loops it is needed to read and write memory and this is programmer’s responsibility. We know when a variable is shared every thread which wants to read it accesses to the same memory location. As in OpenMP there are several threads running at the same time it is totally necessary that all variables within a loop are private. In this situation when a thread uses a variable, it takes its value and saves it in a provisional memory location where only that thread can access. After finishing the loop, this provisional data is deleted.

Next code pieces show the two different ways of doing it.

1. #pragma omp parallel for
2. for (i=0; i < 100; i++)
3. |
4. int temp;
5. temp = array[i];
6. array[i] = do_something(temp);
7. |
#pragma omp parallel for private(temp)

int temp;

for (i=0; i < 100; i++)
{
    temp = array[i];
    array[i] = do_something(temp);
}

In the first example the variable is declared within the parallel construct so it is private by default. On the other hand, in the second example the variable is declared before starting the parallel construct so it would be shared but nevertheless OpenMP pragma defines it as a private variable. The two ways are right but depending on the situation the second one could be more useful.

Continuing with some critical situations we should think about accumulated values. They are not so strange in iterations and they are quite controversial in this case because it needs a shared variable to get the right results but this variable should be private in order to avoid race condition. This is solved by the reduction clause as in the next example.

sum = 0;
#pragma omp parallel for reduction(+:sum)
for (i=0; i < 100; i++)
{
    sum += array[i];
}

By adding the reduction clause basically one private copy of the variable is provided for each thread. Then, after the thread has finished, the value is added to a global variable which sum all the private variables to get the real value. Reduction clause is only compatible with some mathematical (+, -, *, ) and logical (&, |, &&, ||) operations and there can exists multiple reductions in the same loop.

Finally we have to think about how the work division among the different threads in one loop is. Is it equitable? Does it divide the work depending on some parameters? This load balancing is a very important aspect in parallel application performance. When it comes to OpenMP applications it is assumed that all iterations spend the same amount of time. Because of this and the fact that loops use memory sequentially, all threads produce the same number of iterations. This means a good solution for memory issues but not for load balancing issues. The programmer’s task is to deal with the different loop scheduling types which OpenMP provides him/her in order to get an optimal
memory usage and an optimal load balancing. Pragma used in OpenMP for this is as follow:

```
#pragma omp parallel for schedule(kind[, chunk size])
```

Moreover the loop scheduling types are:

- **Static**: Divides the loop into same size chunks
- **Dynamic**: There is a work queue which gives a chunk-size to each thread. Extra overhead is required.
- **Guided**: It is like dynamic but the difference between them is that in this case the chunk-size blocks are large in order to avoid the thread to go to work-queue more than once.
- **Runtime**: It uses `OMP_SCHEDULE` variable to choose one of the three mentioned types.

### 2.3 Non-loop parallelism in OpenMP

Until now we have seen how OpenMP uses parallel clause to split loops across several threads. This means that each time parallel is invoked the threads work starts and when the loop finish, threads are suspended until next parallelization. If there are close parallel sections this can produce overhead as the consumption is higher. The solution which OpenMP offers is using only one parallel section for several loops. An example follows:

```
1. #pragma omp parallel
2. {
3.   #pragma omp for
4.   for (i=0; i<x; i++)
5.     fn23();
6.   #pragma omp for
7.   for (i=0; i<y; i++)
8.     fn24();
9. }
```

In this case the `parallel` clause is invoked only once. Furthermore, it is needed to add one pragma before each loop but this time they will only contain `for` clause. This
solution runs faster than a hypothetic solution with one parallel for clause before each loop because it does not spend time on suspending and restarting threads.

Another interesting construct within OpenMP interface is section. As its name suggests the task of this clause is dividing a piece of code in sections which can be executed by different threads at the same time. Next example shows this:

```c
1. #pragma omp parallel
2. {
3.    #pragma omp for
4.    for (i=0; i<x; 
5.        i++)
6.    Function34();
7.    #pragma omp sections
8.    {
9.        #pragma omp section
10.       TaskA();
11.    }
12.    #pragma omp section
13.       TaskB();
14.    }
15.    #pragma omp section
16.       TaskC();
17.    }
18. }
```

First of all some threads are created to run the loop among them. But now it is different because when the loop finishes the threads are still available instead to be suspended. Next code piece is a group of tasks which could be executed sequentially but it is better to take advantage of the fact that the multiple threads are still running to do it faster. It is achieved by introducing sections clause before the task group and section clause before each task. In this case OpenMP knows absolutely about the thread execution order of the sections (unlike as we saw before in loop scheduling).

Synchronization management is very important in OpenMP. By default, in work-sharing process when a thread finishes its work it waits until all threads have finished their works. However the programmer can change this depending on his/her requirements. Sometimes it would be better for the implementation not make threads to wait for other threads. The clause used in this case is nowait:
CHAPTER 2: OPENMP

1. #pragma omp parallel
2. {
3. #pragma omp for nowait
4. for (i=0; i<100; i++)
5. compute_something();
6. #pragma omp for
7. for (j=0; j<500; j++)
8. more_computations();
9. }

On the other hand a programmer may develop an application where a task cannot be executed until previous tasks are finished. In this case it would be useful to force threads to wait until the end of the tasks. This is achieved thanks to barrier clause:

1. #pragma omp parallel
2. {
3. // First group of tasks
4. #pragma omp barrier
5. //Task which needs that other task have finished to start
6. }

Sometimes the programmer wants parallel sections to be as large as possible in order to reduce overhead. One way to do it is forcing to execute something only once by only one thread. OpenMP has two useful clauses for it. They are single and master. The difference between them is that when single clause is used OpenMP chooses which thread runs the execution. However if the programmer wants to make the master thread to run the execution he/she should use master clause. Next example contains both clauses:

1. #pragma omp parallel
2. {
3. do_multiple_times();
4. #pragma omp for
5. for (i=0; i<100; i++)
6. fn1();
7. #pragma omp master
8. fn_for_master_only();
9. #pragma omp for nowait
10. for (i=0; i<100; i++)
11. fn2();
12. #pragma omp single
13. one_time_any_thread();
14. }

As we can see in the first part of the code there is a function which is called by every thread. Then there is a loop which is divided among the different threads. Although there is no barrier clause, all threads synchronize at the end of the loop. From now on,
the code implements these new clauses. There is a function which can only be executed
by master thread and at the end there is a nowait clause so as soon as every thread
finishes its work each thread will continue to fn2. Finally there is a single clause so next
function will be executed only by one thread (probably the first one that finishes fn2).

When using code in parallel maybe shared-memory synchronization is needed. Because
of this one operation can be interrupted. If the programmer wants to avoid this he/she
must order OpenMP to make that operation as atomic. An atomic operation is never
interrupted and it is achieved by using atomic clause:

1. #pragma omp atomic
2. c[i] += x;

However not every operation can be declared by OpenMP pragma as atomic, this is
only valid for some mathematical and logical operations.

Sometimes two or more different threads want to access to the same code block at the
same time. To avoid this programmer can use critical sections. When a thread has to
execute a critical section, first it ensures that there is not any other thread executing the
section. In this case critical clause is used:

1. #pragma omp critical
2. {
3.   if (max < new_value)
4.     max = new_value
5. }

When critical is used there can be only one critical section. If the programmer wants
more than one critical section in the application he/she must name critical sections. In
this way only the threads that need to block on a particular section will block. Let’s see
a piece of code which shows this:

1. #pragma omp critical(maxvalue)
2. {
3.   if (max <
4.     new_value)
5.     max = new_value
6. }

Apart from the fact that there can be several critical sections, one thread can be at the
same time in several critical sections. However the programmer must be careful with
this because this implementation can produce deadlock.
2.4 Library functions and environment variables

Although pragma clauses are the basis of OpenMP environment there are other elements to take into account. One of them is library functions group. The use of these elements is more complicated than the use of pragmas because it is necessary to make some program changes. The four most common functions are:

- \texttt{int omp\_get\_num\_threads(void)}: Retrieve the total number of threads
- \texttt{int omp\_set\_num\_threads(int NumThreads)}: Set the number of threads
- \texttt{int omp\_get\_thread\_num(void)}: Return the current thread number
- \texttt{int omp\_get\_num\_procs(void)}: Return the number of available logical processors

The two common environment variables are:

- \texttt{OMP\_SCHEDULE}: Controls the scheduling of the for-loop work-sharing construct.
- \texttt{OMP\_NUM\_THREADS}: Sets the default number of threads. The call can override this value.

2.5 Conclusions of OpenMP

We have seen that OpenMP provides programmers an easy way of implementing parallelization. To sum up OpenMP, the main features are as follows:

- **Standardization**: It provides a standard among a variety of shared memory architectures/platforms.
- **Efficiency**: It establishes a simple and limited set of directives for programming shared memory machines. Significant parallelism can be implemented by using just 3 or 4 directives.
Ease of Use: It provides capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach. Provide the capability to implement both coarse-grain and fine-grain parallelism.

Portability: It supports Fortran (77, 90, and 95), C, and C++.
CHAPTER 3

MPI (MESSAGE PASSING INTERFACE) [4], [5], [6], [7], [8]

3.1 What is MPI?

MPI (Message Passing Interface) is a standardized interface for performance of parallel applications based on message passing. The programming model that underlies MPI is MIMD (Multiple Instruction streams, Multiple Data streams) although there are special facilities for using the SPMD model (Single Program Multiple Data), a particular MIMD in which all processes running the same program, although not necessarily the same instruction simultaneously.

MPI is an interface so the standard does not require a particular implementation. It is important to give a collection of functions to the programmer in order to design his application, without necessarily knowing the hardware or the implementation of the functions.

Figure 1: Location of MPI in parallel applications programming.
Its main feature is that it does not need shared memory so it is a very important option when it comes to work with distributed systems. The three main elements in MPI are the delivery process, the receiver process and the message.

MPI has been developed by the MPI Forum, a group of researchers from universities, laboratories and companies involved in high-performance computing. The fundamental objectives of MPI Forum are:

- Defining a unique programming environment that ensures portability of parallel applications.
- Defining the programming interface absolutely with no information about the implementation.
- Providing public good-quality implementations in order to improve the standardization.
- Convincing the parallel computer manufacturers to offer optimized MPI versions for their machines (some manufacturers such as IBM and Silicon Graphics have already done it).

The basic elements of MPI is an programming interface non-related to any language and a collection of bindings or concretions for the most common programming languages in parallel computing community: C and FORTRAN. However MPI has been implemented in other languages such as C++, Ada or Python.

A developer who wants to use MPI for his projects will work with a concrete implementation of MPI. It consists of at least these elements:

- A library of functions for C and the header file \texttt{mpi.h} with the definitions of those functions and a collection of constants and macros.
- Commands for compilation, typically \texttt{mpicc}, \texttt{mpif77}, which are versions of the usual compilation command (\texttt{cc}, \texttt{f77}) which incorporate MPI libraries automatically.
- Commands for the execution of parallel applications, typically \texttt{mpirun}.
- Tools for monitoring and debugging.
As we saw before, MPI is specially designed for SPMD application development. When an application starts, it is released into N parallel copies of the program (processes). These processes don't progress synchronized in each instruction, the synchronization (when it is necessary) must be explicit. The processes have a completely separate memory space. Information and synchronization exchanging are done by passing messages.

There are available point to point communication functions (involving only two processes) and group functions or operations (involving implies multiple processes). Processes can be grouped as communicators allowing a definition of the scope of collective operations as well as a modular design.

The structure of a typical MPI program, using the bindings for C, is the following:

```c
1. # include "mpi.h"
2. main (int argc, char **argv) {
3.     int nproc; /* Number of process */
4.     int yo; /* My address: 0≤yo≤(nproc-1) */
5.     MPI_Init(&argc, &argv);
6.     MPI_Comm_size(MPI_COMM_WORLD, &nproc);
7.     MPI_Comm_rank(MPI_COMM_WORLD, &yo);
8.     /* PROGRAMM SKELETON */
9.     MPI_Finalize();
10. }
```

Next code segment presents four of the most common functions in MPI: `MPI_Init()` to start the parallel application, `MPI_Comm_size()` to determine the number of processes involved in the application, `MPI_Comm_rank()`, so that each process determines its address (identifier) in the collection of processes in the application, and `MPI_Finalize()` to finish the application.

```c
1. int MPI_Init(int *argc, char **argv);
2. int MPI_Comm_size (MPI_Comm comm, int *size);
3. int MPI_Comm_rank (MPI_Comm comm, int *rank);
4. int MPI_Finalize(void);
```

The above example also shows some MPI conventions. The names of all functions begin with "MPI", the first letter is always capitalized and the rest are case sensitive also.

Most MPI functions return an integer which is a diagnosis. If the return value is `MPI_SUCCESS`, the function is successful. No other possible values are standardized.
The key word `MPI_COMM_WORLD` refers to the universal communicator which is predefined by MPI and includes all application processes. All MPI communication functions need a communicator as argument.

### 3.2 Point-to-point communication

There are a number of MPI functions which allow communication between two specific processes. It depends on the model and the communication way.

**Different models and communication ways**

MPI defines two communication models: blocking and nonblocking. Communication model is related to the time a process is blocked after calling some communication function (delivery or reception). In a blocking function the process is blocked until the end of the requested operation. On the other hand if the function is nonblocking the operation is requested but the control is returned immediately without waiting for the end. In this case the process must check later if the operation has finished or not.

At this point the reader may think about when an operation is finished. If it is a reception it is clear that it is when the message is received and stored in the buffer. However if it is a delivery we could have some doubts. In MPI a delivery is finished when the sender can use the buffer again without any problem which was used for sending the message.

When it comes to communication ways MPI offers programmer four ways of communication:

- **Basic**: Its functionality depends on the implementation. If the messages are light buffer is used but if the messages are quite large synchronization replaces buffering.

- **Buffered**: A copy of message is saved immediately in the buffer. Operation is finished when this copy is already done. If buffer space is not enough the operation fails.
• **Synchronous:** Operation is finished only when message is received by the destination.

• **Ready:** This way is only useful if the other side of the communication is ready for an immediately transfer. There is any copy of the message.

**Basic communication**

As the result of two models and four communication ways we saw before we obtain eight delivery functions. However there are only two reception functions.

`MPI_Send()` and `MP_Recv()` are basic delivery and reception blocking functions:

```c
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm,
             MPI_Status *status);
```

Let’s see what each element means:

• **Buf, count and datatype:** They set up the message which is sent or received.

• **Dest:** It is the identifier of the receiver process.

• **Source:** It is the identifier of the process which is expected to send the message.

• **Tag:** In `MPI_Send()` it is mandatory but in `MP_Recv()` is optional.

• **Comm:** It is a communicator. It is usually the universal communicator (`MPI_COMM_WORLD`).

• **Status:** Its task is informing about different statuses of the message received.

On the other hand `MPI_Isend()` and `MPI_Irecv()` are basic delivery and reception nonblocking functions:

```c
int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,
              MPI_Request *request);
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm,
              MPI_Request *request);
int MPI_Wait(MPI_Request *request, MPI_Status *status);
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
int MPI_Cancel(MPI_Request *request);
```
In this case there is a new element: request. This is a requested operation receipt. It could be used for checking later if the operation has finished or not. *MPI_Wait()* function takes a receipt and it block the process until the end of the current operation. Because of this, we can say that executing *MPI_Wait()* just after *MPI_Isend()* is like executing *MPI_Send()*). If programmer only wants to know if the operation has finished but not to block the process the ideal function is *MPI_Test()*). If the flag is 1 the function has finished.

Finally *MPI_Cancel()* function allows to cancel an unfinished operation.

**Buffered communication**

One of the disadvantages that basic communication has is that programmer does not have control over the operation length. Sometimes it could be a problem because the process may be blocked for too long time.

In order to avoid this, programmer can use a buffer to store a copy of the message. This way there is no risk of deadlock while the other side is expected to be ready. In this case the function is *MPI_Bsend()*:

```
int MPI_Bsend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);
```

As we can see it is almost the same as *MPI_Send()*). Nevertheless it is needed some functions more when programmer works with buffered communication. He/she has to set a buffer by using *MPI_Buffer_attach()*:

```
int MPI_Buffer_attach(void* buffer, int size);
```

When the process has finished and the receiver has got the message the buffer needs to be empty again. This is achieved by using *MPI_Buffer_detach()*:

```
int MPI_Buffer_detach(void* buffer, int* size);
```

If the buffer size is not enough for the message the process will fail. In this case the programmer will have to change to a bigger buffer or change to another communication way where buffer requirements are lower. Note that in the second case the deadlock risk would be higher.
3.3 Collective operations

Many applications need communication between more than two processes so point-to-point communication is not valid. For this reason MPI provides a set of collective operations:

- Synchronization barriers
- Broadcast
- Gather
- Scatter
- Mathematical operations (sum, multiplication…)
- Combination of previous operations

A collective operation must be invoked by all partners although their roles are not the same. Most of these operations need to declare a root process.

**Barriers and broadcast**

*MPI_BARRIER*() does not need any information exchange. It is basically a synchronization operation which blocks the processes until all of them have crossed the barrier. It is usually used to set a program as finished before starting with other one.

```c
int MPI_BARRIER(MPI_Comm comm);
```

*MPI_BROADCAST*() is used by the root process to send a message to all communicator processes. This function fulfills one of the main features of collective operations: all processes invoke the same function and declare the same process as the function root.

```c
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

**Gather**

Function used for gather is *MPI_GATHER*(). It collects data in root process. Data are stored consecutively.
CHAPTER 3: MPI (MESSAGE PASSING INTERFACE)

int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);

recvbuf, recvcount and recvtype are only necessary in the root process but sendbuf, sendcount and sendtype are necessary in all processes (include root).

If programmer wants to store data not consecutively he must use MPI_Gatherv(). Moreover each process has different blocks size.

int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm);

recvcounts table is used for storing the size of every process and displs shows the displacement between two blocks.

Scatter

MPI_Scatter() operation is the opposite operation than MPI_Gather(). First the root process has an elements vector (one per process) and then it distributes a copy of that vector to every process. Note that MPI allows sending not only individual piece of data but also data blocks.

int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);

In this case if data is not stored consecutively it is needed to use MPI_Scatterv(). Its interface is more complex than in MPI_Scatter() but it is more flexible.

3.4 Data types

MPI has a collection of simple data types but also derivate data types.

Simple data types

MPI defines a collection of data types for each language supported. As most used language related to MPI is C we will see some examples from C collection:
### Chapter 3: MPI (Message Passing Interface)

<table>
<thead>
<tr>
<th>MPI types</th>
<th>Equivalent C types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>No equivalent</td>
</tr>
</tbody>
</table>

A C application needs to provide to MPI a data type equivalence table when it starts a message. The reason of this is that not all processors used for a MPI application may be the same; it could be possible that there are differences between the ways of representing data so MPI would change syntaxes in order to make possible working in heterogeneous environments.

#### Derivate data types

MPI data transfer usually works with simple data types. However sometimes it is necessary to transfer structured types or vectors. MPI is able to define more complex types by using constructors. Before using a user type it is needed to execute `MPI_Type_commit()`. When it is not needed it can be released by using `MPI_Type_free()`.

```c
int MPI_Type_commit(MPI_Datatype *datatype);
int MPI_Type_free(MPI_Datatype *datatype);
```

The function used to define them depends on if they are homogeneous or heterogeneous. If they are homogeneous (all elements have the same type) function is `MPI_Type_contiguous()`. On the other hand, if they are heterogeneous (different types) function is `MPI_Type_struct()`.

```c
int MPI_Type_contiguous (int count, MPI_Datatype oldtype, MPI_Datatype *newtype);
MPI_Type_struct (int count, int *array_of_blocklenghts, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype);
```
3.5 Conclusions of MPI

In this chapter we have seen Message Passing Interface, an API which defines the syntax and the semantic of a library used in several languages to run parallelization in multi-processors systems. The main features of MPI are:

- Point-to-point communication
- Data types
- Process groups
- Communication contexts
- Process topologies
- Environmental Management and inquiry
- The info object
- Process creation and management
- One-sided communication
- External interfaces
- Parallel file I/O
- Language Bindings for Fortran, Ada, Python, C and C++

Finally we will see the typical “hello world” example implemented in C by using MPI. Program sends a message to each process and then it is printed. Each process manipulates the message and sends the results to the main process.

1. #include <mpi.h>
2. #include <stdio.h>
3. #include <string.h>
4. #define BUFSIZE 128
5. #define TAG 0
6. int main(int argc, char *argv[])
7. {
8.     char idstr[32];
9.     char buff[BUFSIZE];
10.    int numprocs;
11.    int myid;
12.    int i;
13.    MPI_Status stat;
14. `MPI_Init(&argc, &argv); /* all MPI programs start with MPI_Init; all 'N' processes exist thereafter */`

15. `MPI_Comm_size(MPI_COMM_WORLD, &numprocs); /* find out how big the SPMD world is */`

16. `MPI_Comm_rank(MPI_COMM_WORLD, &myid); /* and this processes' rank is */`

17. /* At this point, all programs are running equivalently, the rank distinguishes the roles of the programs in the SPMD model, with rank 0 often used specially... */

18. if(myid == 0)

19. {

20.   printf("%d: We have %d processors\n", myid, numprocs);

21.   for(i=1;i<numprocs;i++)

22.   {

23.     sprintf(buff, "Hello %d! ", i);

24.     MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);

25.   }

26.   for(i=1;i<numprocs;i++)

27.   {

28.     MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD, &stat);

29.     printf("%d: %s\n", myid, buff);

30.   }

31. }

32. else

33. {

34.   /* receive from rank 0: */

35.   MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat);

36.   sprintf(idstr, "Processor %d ", myid);

37.   strncat(buff, idstr, BUFSIZE-1);

38.   strncat(buff, "reporting for duty\n", BUFSIZE-1);

39.   /* send to rank 0: */

40.   MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);

41. }

42. 

43. `MPI_Finalize(); /* MPI Programs end with MPI Finalize; this is a weak synchronization point */`

44. return 0;

45. `}`
CHAPTER 4

CODEPLAY SIEVE SYSTEM \cite{9, 10, 11}

4.1 What is Sieve?

Sieve C++ Parallel Programming System was designed by Codeplay. It is a C++ compiler and parallel runtime whose aim is simplifying the parallelization of code in order to work with multi-processor systems.

The piece of code where the programmer wants to perform parallelization is marked with the sieve marker. Inside this piece of code (sieve blocks) side effects are delayed, the sieve blocks are split in three parts: reading from memory, computation and writing to memory. Due to this, parallelization is performed in the program compilation and the memory operations are separated.

When it comes to work with Sieve the programmers must be aware of the three main rules:

- A sieve is a block of code contained within a sieve {} marker and any marked functions by sieve.
- All side-effects are delayed until the end of the sieve.
- One side effect is a modification of data that was declared outside the sieve block.
Let’s see a simple example of Sieve piece:

```c
1. void simple_loop (float *a, float *b, float c, int aux) {
2.   sieve {
3.     for (int i=0; i<aux; i++) {
4.       a[i] = b[i] - c;
5.     }
6.   } // the assignments to the 'a'
7.   array are delayed until here
8. }
```

There is a C++ loop where the pointers `a` and `b` can be pointing to outside memory spaces so sieve is used to parallelize the loop because it makes that all assignments to `a` are being delayed until the end of the loop.

Another important concept in Sieve technology is dependency. It is a situation where one piece of the program must be executed after another one. At first sight this clashes with parallelization because it is needed to change the order of execution. This is solved by delaying the side effects. It means removing a lot of dependencies so the compiler can change the order of the execution without altering the right functionality of the program.

Note that dependencies only can exist on local variables inside a sieve block. Other elements such us global variables or pointers to external data cannot have any dependency. However there are still some dependencies and they should be removed in order to gain a total parallelism so the programmer have to choose how to remove these last dependencies. Then the compiler will be able to auto-parallelize.

Due to working with data outside the sieve and data inside the sieve multiple memory spaces are needed. This makes multi-core programming experience better because each processor has its own memory space and it can load and store data in a fast way by using its local memory.

There is a special feature to achieve transferring data between the different memory spaces quickly which is called Direct Memory Access (DMA). Note that this is a general and important feature not only in multiprocessor programming area.

Sometimes the algorithm must change to get parallelism so the programmer needs a way to specify this. This can be solved by going through an array of integers and adding
up the numbers to produce the total. However this is valid for single-core programming but not in multi-core programming because if the total variable was inside the sieve block it would not be considered as a side-effect so the total would not be the correct.

Sieve has a solution for this too. It consists special library classes that can be used with local variable. Let’s see an example of this:

```
int sum (int *array, int size) {
    int result;
    sieve {
        IntSum total (0) splits result;
        IntIterator index;
        for(index=0;index<size;index++) {
            total += array [index];
        }
    }
    return result;
}
```

First, `IntSum` class is used for getting parallel accumulators. If the programmer wants to split the sum operation into across multiple processors he/she can use several methods which are defined in the class definition.

On the other hand we have to pay attention to `IntIterator`. It splits the variable and send it to different processors of the system.

One important programming element which must be taken into consideration is functions. If the programmer called some function inside the sieve block it could cause side-effects. Due to this the normal C++ function calls are delayed until the sieve block execution is finished. The programmer can use the normal C++ functions without any worry but he/she has to be aware about that the results of these functions will be launched at the end of the block. But if the programmer needs to call some function immediately within the sieve block he/she must declare the function as a sieve function by adding the word “sieve” at the end of the function definition. This kind of functions delay all their side-effects like sieve blocks. Owing to this, sieve functions can safely be called in parallel. Sieve declares functions as follows:

```
int function (int parameter) sieve;
```
What happens if a sieve function modifies main memory? It is not a problem because the modifications are delayed until the end of the sieve block.

The last element we analyze is the “immediate pointers”. They allow programmer to create complex data structures within sieve blocks. They can write to data pointed with an immediate effect. The main advantage of this is that these complex structures use the local memory instead the main memory so the process is faster. They are defined as follows:

\[ int \ast 0\ p; \]

### 4.2 Implementation of Sieve

First of all we can see in Figure 2 the “implementation way”. 
Before seeing the real process we should know about how the compiler works. The side-effects caused inside sieve blocks are converted into delayed side-effects. In order to get auto-parallelization the compiler tries to find the dependencies in the sieve blocks. Note that in this case these delayed side-effects could be calculated in any order but at the end they are executed in the right order so there is not any read-after-write dependency pointed to data outside the sieve block. With this, we obtain a simpler analysis because it only needs to be performed on variables inside the sieve block.

So the aim of the compiler is finding points without any dependencies. These points would be called as split point (we can see them in the figure). However they could not be found without using split/merge operations. If with this the split points are not found we can say that the auto-parallelization is not safe in this case.

**Figure 2: Implementation of Sieve**
Other important elements in Figure 2 are the fragments. They start or end in some split points. They have one entry point and one or more exit points. When some of these fragments are “brothers” they can be executed in parallel. However, not always the best option is the maximum parallelization. If there are a lot of parallelism there is more cost of calling split/merge operations so each architecture has an optimal size of the fragments.

As there are fragments with more than one exit point the run-time determine which fragment is executed next. But if the run-time executes a fragment where there are several exit points it has a little problem because it does not know in advance which exit point the fragment will choose. In this situation there are two possible ways. The first one is executing speculatively and the other one is waiting until the end of the fragment and execute the fragment which corresponds to the exit point it did choose. The performance of the speculatively option is not so much efficient but it increases parallelism so it improves the performance.

4.3 Conclusions of Sieve

We have seen Sieve compiler and now it allows a programmer to set auto-parallelization. In this case it can only be implemented in C++. However it is a very easy way to that implementation in several parallel processors:

- Multi-core special-purpose processors.
- Special-purpose coprocessors.
- Dual-core and quad-core processors.
- Multiple Processor server systems.
CHAPTER 5

CHAPEL [12], [13], [14], [15]

5.1 What is Chapel?

Chapel is a new parallel programming language designed by Cray. Its name comes from Cascade High-Productivity Language. As its name says it is part of Cascade, a project by High-Productivity Computing Systems program. The aim of it is to develop an advanced high-performance computing system. This program was created because, according to its developers, programming new multi-core supercomputers becomes more and more difficult.

Chapel takes many language features from modern programming languages and mixes them with high-performance parallel language abstractions. Chapel considers a machine as a parallel computing system with a specific number of homogeneous processors. As Chapel is an explicitly parallel programming language, it is programmer’s responsibility for getting the available concurrency in code.

Chapel is an independent language not a library for another language or an API so it would be too heavy for this thesis to see all data types, variables, functions, etc… However elements of Chapel Syntax are very similar to the elements of Java, C, FORTRAN or Ada. Chapel is not only a set of clauses or a library, but it is a complete language so we will not go deeper into language specifications because it would change the subject of this thesis. Moreover it is maybe the less common technology researched in this thesis.
5.2 Chapel principles

Chapel language is based on four principles:

- General parallel programming
- Locality-aware programming
- Object-oriented programming
- Generic programming

The aim of general parallel and locality-aware programming principles is supporting general, performance-oriented parallel programming with high-level abstractions. On the other hand, the aim of object-oriented and generic programming is reducing the width of high-performance parallel programming languages, mainstream programming, and scripting languages.

**General Parallel Programming**

The most important thing to know is that Chapel has been designed to support parallel programming by using high-level language abstractions. Chapel works with a global-view programming model that increases the abstraction of expressing data and control flow when they are compared to parallel programming models used in production.

Global-view data structures are, for example, arrays. However, they can be other kinds of data aggregates. Although some of their implementations distribute them across locales of parallel systems, their sizes and indices are expressed globally. A locale means a unit of uniform memory access on target architecture. So inside a locale, all threads have similar access times to the same memory address.

This global view of data is different from the majority of parallel languages which make users smash distributed data aggregates into per-processor chunks instead of doing manually or with language abstractions. A global view of control is the fact that a user’s program starts the execution with a single logical thread of control and then adds to the process additional parallelism.
Multithreading is very important in Chapel because Chapel implements all parallelism by it though these threads have been created with high-level languages and have been managed by the compiler and runtime, instead of explicit fork/join-style programming.

As a result of the former Chapel can express parallelism in a more general way than the SPMD model (Single Program Multiple Data) that most common parallel programming languages use in their execution and programming models. However if the programmer still wants to use the SPMD model way can do it in Chapel without any problem.

Another consequence of general parallel programming is that a broad range of parallel architectures is targeted. One of Chapel’s aims is targeting a wide spectrum of HPC hardware with some elements such as vector, multi-core processors, multithreading and custom architectures such as distributed-memory or shared-memory. The goal of Chapel’s developers is having Chapel program run correctly on all of these architectures.

**Locality-Aware Programming**

Another Chapel’s principle is to give the chance to the user to specify where computation and data should be stored on the physical machine. This is very important in order to make scalable performance on large machines. This is different than in shared-memory programming models where the user uses a flat memory model. It also contrasts with SPMD-based programming model where these details are specified by the programmer on a process-by-process basis with the multiple cooperating program instances.

**Object-Oriented Programming**

Support for object-oriented programming is other important principle in Chapel. By using object-oriented programming the productivity in the mainstream programming community has increased because of its encapsulation of functions and related data into a single software component. It supports specialization and reuse and they are used as a clean mechanism to define and implement interfaces. Of course objects are allowed in Chapel so it has the benefits of an object-oriented language. Chapel works with traditional reference-based classes and with value classes.
It is not needed for the programmer to work with an object-oriented manner. Due to this, FORTRAN or C programmers do not have to change their way of programming paradigm if they want to use Chapel effectively. However, some of Chapel’s standard library capabilities have been implemented by using objects so the programmer could have to use a method-invocation style of syntax in order to be able to work with those capabilities.

**Generic Programming**

The last Chapel’s principle is supporting for generic programming and polymorphism. Thanks to these features the code can be written in a generic style. In this case the aim is supporting exploratory programming and scripting languages and supporting code used before. This flexibility is achieved by making the compiler to create versions of the code for each required type signature.

### 5.3 Conclusions of chapel

We have taken an overview of a language completely designed for parallel programming purposes. In this case developers choose to create it from scratch instead creating a library for an existing language or implementing an API which manages parallelization. Learning this language could be a good option if future projects are going to be developed by multi-core systems and parallelization is required. However as we are seeing in this thesis there are other easier options for implementing parallelization so chapel is only the best in specific situations. Moreover chapel is a new language and it is not as common as other technologies so there is not so much support for it.
CHAPTER 6

CILK++ \cite{16}, \cite{17}, \cite{18}, \cite{19}

6.1 What is Cilk++?

Cilk is considered as a general-purpose programming language for multithreaded parallel computing. Actually it is an extension to well-known language C which provides an easy, quick and reliable way to improve the performance when it comes to work with C programs on multi-processors systems. It was created at MIT Laboratory in 1994.

In 2007 Cilk was relegated by a modern version of it, Cilk++. The differences between them are that Cilk++ is based on C++, supports loops, operation with Microsoft and GCC compilers and Cilk hyperobjects.

Later in 2009 Cilk were bought by Intel. Due to this a new version of Cilk was launched in September 2010 as Intel Cilk Plus. This new version differs from previously version by being integrated in a commercial compiler, adding some array extensions and finally the compatibility with many debuggers have been improved.

Focusing on Cilk++, it is a very good option if cases of divide and conquer algorithms. Recursive functions are usually managed by these algorithms so they are very well supported by Cilk++.
6.2 Cilk++ working model

As the purpose of this thesis is researching about parallelization implementations we divide code pieces into serially execution and pieces where maybe it is executed in parallel. These code sections where there are not any parallel control structures are called strands.

Other important concept is knot. It is a point where two or more strands converge in only one or one strand is divided into several strands. The first one is a spawn knot and the second one is a sync knot. Next figure shows these concepts:

![Figure 3: Cilk++ execution runtime diagraph A.](image)

In this case the piece of code would be as follow:

```cpp
do_stuff_1(); // execute strand 1
cilk_spawn func_3(); // spawn strand 3 at knot A
do_stuff_2(); // sync at knot B
do_stuff_4(); // execute strand 4
```

In this figure, called DAG (Directed Acyclic Graph), both strands (1-4) and knots (A-B) are represented. Note that A is a spawn knot and B is a sync knot.

Nevertheless one strand can be executed in parallel with first one strand and then others strands as Figure 3 shows:
In this program while strand 3 is been executed, first strand 2 and then strands 4 and 5 (in parallel between them too) are executed.

However this is a very simple way of representing a parallel structure of a Cilk++ program because it does not take into consideration the number of processors or the duration of each strand. In order to solve this another concept is added. Work is the total amount of processor time which a program needs to be completed. Each strand has its own work value as we can see in example below.

Another important concept is span. It is the longest path in which needs more time to be completed. In this DAG work would be 181 milliseconds and span would be 68 milliseconds. It means that in a hypothetical situation where there is unlimited number of processors the duration of program execution should be 68 milliseconds.

If we move to mathematical scope and we assume that \( T(P) \) is the execution time of the program by using \( P \) processors, the work and span will be:
- Work: $T(1); T(P) \geq T(1)/P$
- Span: $T(\infty); T(P) \geq T(\infty)$

Moreover two concepts are defined more, speedup and parallelism.

- Speedup: $T(1) / T(P)$
- Parallelism: $T(1) / T(\infty)$

These concepts can be very useful to predict the speedup of a program. If $T(1)$ and $T(\infty)$ values are known this prediction can be achieved. For this purpose cilkview was implemented. It makes an estimate about speedup length by measuring these values and others and allow programmer to know more about the behavior of the program.

Finally it is important to know that DAG does not depend on the number of processors.

### 6.3 Cilk++ execution model

In Cilk++ environment when a strand is declared after a spawn node it is not required to run in parallel. Of course it could be executed in parallel but it is not mandatory. This decision is made by the scheduler. In Cilk++ context scheduler does it by work-stealing.

If we take first DAG example and consider the following Cilk++ piece of code:

```cpp
do_init_stuff();  // execute strand 1
cilk_spawn_func3();  //spawn strand 3 (“child”)
do_more_stuff;  //execute strand 2 (“continuation”)  
cilk_sync;
do_final_stuff;  //execute strand 4
```

The program can be executed in two different ways. It can be executed totally by only one worker or some strands (in this case 2 and 3) can be executed by different workers. A worker is a system thread that executes a program. The child strand is always executed by the same worker as the strand before the spawn knot but if there is any other worker available strand will be executed by it instead by the “main” worker. This is the basis of work-stealing concept. Next figure illustrates this:
We know strands 1 and 3 are executed by “main” worker and strand 2 is executed by other alternative worker which is available in that moment but what happens with strand 4? We cannot know in advance which worker will execute it because the worker which does it will be the last to reach the sync.

### 6.4 Conclusions of Cilk++

Like Chapel, Cilk++ is an entire language designed for parallelism purposes. However the conclusions of them are much different because in fact Cilk++ is not a real language but also an extension to a common OOP language, in this case C or C++ (depending on if it is Cilk or Cilk++). Moreover it is more intuitive and easier to learn than Chapel because maybe programmer does not need to learn about variables, functions, etc… It is a good option for C++ programmers who want to implement parallelization in applications. On the other hand the fact that decision of which processor runs each strand is quite controversial because it makes programmers not to worry about it but takes out managing capabilities from programmers.
CHAPTER 7

SKANDIUM [20], [21]

7.1 What is Skandium?

Skandium is a Java library for multi-core architectures. It works in multi-core context at high-level so the parallelism is the only way to provide a faster execution. It achieves this by using some parallelism patterns such as farm, pipe, for, while, if, map...

Skandium has been developed by NICLabs, part of the Department of Computer Science (DCC) of the University of Chile. Actually it is a total re-implementation of Calcium, another library which is part of ProActive Parallel Suite which works with distributed architectures such as Grids and Clusters.

Skandium is based on Algorithmic Skeletons technology which traditionally has been oriented to distributed systems as we saw before and it uses thousands of node calculation.

7.2 Skandium Programming Model

Skeletons

The programming model’s construct consist of two different categories. First one is skeleton patterns which represent different patterns of parallel computation. All details about communication are implicit for each pattern so the programmer does not know them. The different skeletons are:
- **Farm**: It is also known as master-slave. Nested patterns within farm are used in order to get parallelism.

- **Pipe**: Parallelism is reached by executing different stages at the same time for different tasks.

- **If**: It provides conditional ramification. By using a condition one sub-skeleton is executed dynamically.

- **For**: It executes a sub-skeleton a predefined number of times.

- **While**: It executes a sub-skeleton a dynamical number of times depending on a condition.

- **Map**: Each task is divided into sub-tasks. Then each sub-skeleton is executed by all sub-tasks at the same time. Finally all results are put together.

- **Fork**: In this case each sub-skeleton is executed by only one sub-task. Then the results are merged.

- **D&C**: It is similar to Map but in this case it is executed recursively meanwhile a condition is fulfilled. Each sub-task decides if it executes the sub-skeleton or not depending on that condition.

**Muscles**

In order to transform a skeleton into an application programmer must “fill” the skeleton with specific functions from his/her application. These functions are called as muscles. Muscle methods are probably executed concurrently. Because of this if the implementation of the method is not stateless it should be very careful with the synchronized keyword. Moreover synchronization could disable parallelism in some skeletons so it should be avoided as much as possible.

Muscles are grouped into 4 types:

- **Execute**: public R execute (P param); it takes one entry argument and it returns one argument.
CHAPTER 7: SKANDIUM

- **Split**: `public R [] split (P param);` it takes one entry argument and it returns one results list.

- **Reduction**: `public R reduction (P[] param);` it takes one arguments list and it returns only one result.

- **Condition**: `public Boolean condition (P param);` it takes one argument and it returns true or false.

Muscles are executed in a parallel and concurrent way. Muscle’s results could be other muscle’s parameters. Execution order of muscles is defined by skeleton related to them. Hence, it is very important that programmer knows muscles will be executed concurrently.

**Example: Pi**

Next example calculates the first 3000 decimals of Pi by using Bailey-Borwein-Plouffe formula. Best skeleton pattern for this is Map because it divides a problem into sub-problems, calculate each of them in a individual way and finally reduce the results.

Four steps are required for using library:

1. Define the skeleton and its respective muscles. In this case it is a map skeleton.

   ```java
   Skeleton<Interval, BigDecimal> bbp = new Map<Interval, BigDecimal>(
       //number of parts to divide by
       new SplitInterval(Runtime.getRuntime().availableProcessors()*2),
       new PiComputer(),
       new MergeResults());
   ```

2. Insert data to library, in this case an interval of decimals [0, 3000]. When interval is inserted a `future` variable is created to store the result when ready. Multiple data can be inserted but in this case it is only one.

   ```java
   Future<BigDecimal> future = bbp.input(new Interval(0, 3000));
   ```

3. Meanwhile main thread can do other tasks.

4. Obtain results from `future` or block it until they are available.

   ```java
   BigDecimal pi = future.get();
   System.out.println(pi);
   ```
7.3 Conclusions of Skandium

At first sight the main advantage of Skandium is that it makes parallelization possible in Java applications. Most of other technologies related to parallelization and multi-processor working usually are implemented in other languages such us FORTRAN, C or C++. Moreover it is not difficult way to implement it. It is true that there are easier ways of implementing but we cannot say Skandium is a very complex library. Programmer is only supposed to know about different skeleton patterns library provide him/her and think about how to fill it with his/her functions (muscles). To sum up Skandium is a very interesting Java library which targets parallel programming of multi-core architectures.
CHAPTER 8

SOME COMPARISONS BETWEEN PREVIOUS PROGRAMMING TECHNOLOGIES

8.1 Technology Comparison Table

Once we have seen the most important aspects of these common technologies related to multi-core programming we should compare them and try to find in which situations each of them are more suitable although it is difficult to compare them because they are very different technologies, some of them are API, others are languages,…

First we will see a table with a comparison between some general features of all of them:
8.2 Chapel limitations and Cilk++ vs. Skandium

As in some cases they are so different technologies the environment where each of them are a good way to implement parallelization is different. At first sight the most different technology is maybe Chapel. Although Cilk++ is a language too, the fact is that Cilk++ is an extension for C++. It is true that Chapel is not very different than other common languages such as Java but it requires user to learn a new language. Due to this important disadvantage we can say that Chapel is the less useful technology presented in this thesis. However there could be some situations where it is a good option because it allows user to manage parallelization in a more complex and customizable way. It could be the best option if the programmer wants to develop an application where parallelization is constantly required and it is needed to work in a specific way.

Leaving Chapel behind Cilk++ and Skandium show some similarity between them. Both are an extension or library for a common language so as to achieve parallelism. Skandium supports Java and Cilk++ supports C++. As we can see in the previous table Java parallelization support is more limited than C++ support. This fact makes Skandium a technology to take into account. Moreover it offers user more
parallelization flexibility than Cilk++ do. However Skandium is developed by a university and there is not many community supports for it. On the other hand Cilk++ is supported by Intel so it is easier to find documentation and support about Cilk++ than about Skandium. When it comes to memory aspects Cilk++ uses Miser memory manager, a better option than shared memory system used in Skandium. Furthermore in Cilk++ environment overhead is quite small when compared with other pthread libraries. To sum up Cilk++ is a better option than Skandium in global terms. However the fact that it needs a custom compiler and it does not support the very common language Java can make some programmers to reject it and choose Skandium.

8.3 The “Big Three”: OpenMP, MPI and Sieve

After that we have to review and compare OpenMP, MPI and Sieve System. OpenMP and MPI are the most common and, except some situations, the most useful technologies in this thesis. Sieve differs from them especially from MPI so first we will compare OpenMP and MPI and then we will make a comparison between Sieve and OpenMP.

It is not totally clear between OpenMP and MPI which one is better. Both have some advantages and disadvantages:

**OpenMP advantages**

- Easier to program and debug than MPI.
- Directives can be added incrementally – gradual parallelization.
- The program can still be run as a serial code.
- Serial code statements usually don't need modification.
- Code is easier to understand and maybe more easily maintained.

**OpenMP disadvantages**

- It can only be run in shared memory computers.
- It requires a compiler that supports OpenMP.
- It is mostly used for loop parallelization.
CHAPTER 8: SOME COMPARISONS BETWEEN PREVIOUS PROGRAMMING TECHNOLOGIES

MPI advantages

- It runs on either shared or distributed memory architectures.
- It can be used on a wider range of problems than OpenMP.
- Each process has its own local variables.
- Distributed memory computers are less expensive than large shared memory computers.

MPI disadvantages

- It requires more programming changes to go from serial to parallel version.
- It can be harder to debug.
- Performance is limited by the communication network between the nodes.

In my view OpenMP is better for beginner programmers because it is easier than MPI. Moreover one of OpenMP specialties is loop parallelization so if programmer needs some technology for this purpose OpenMP will be the best option for him/her. Although OpenMP is very common and very useful in many situations it has limitations as it can only be run in shared memory computers. This fact makes MPI the best option when computer used cannot offer shared memory and each process and variables need to be stored in its own memory space instead of shared memory.

Finally we should take into consideration Sieve System because it is a simple but very efficient solution and it could be the “surprise” of the thesis. Of course it has some disadvantages as it does not allow a very complex parallelization way and it only supports C++ but it can be the best option when it comes to implement parallelization in an easy and simple way. It has some advantages comparing to OpenMP:

- In OpenMP the programmer needs to provide detailed and correct information about the C++ program which he is working with. This information is needed by OpenMP parallelizer in order to achieve a right performance. If not, parallelizer would produce a program with undefined behavior and this produces unreliability.
- The OpenMP parallelizer cannot check that the information given by the programmer is correct; meanwhile the Sieve C++ parallelizer can check
dependency information. Moreover it can launch clear errors and some advice about making the program more parallel.

- In Sieve C++ the programmer can debug the programs in a single-threaded environment.
- The Sieve C++ system is able to manage multiple non-uniform memory spaces. Data is separated in memory spaces by the sieve blocks. One important advantage is that variables within sieve blocks are located in memory spaces close to the processors and variables and heap data are in the largest shared memory space. Even it is possible to manage multiple hierarchical memory spaces.
- Sieve C++ system can work with DMA and it can load and store memory architectures.

As the only requirement for working with Sieve is its own compiler, next appendix will try to set up the other two most important technologies in this thesis: OpenMP and MPI.
APPENDIX A

INSTALLATION AND CONFIGURATION OF OPENMP AND MPI FOR C++

In this appendix we are going to see an explanation about how to install and configure the two most relevant technologies explained in this thesis: OpenMP and MPI. We will work on Linux environment and we will use Synaptic and KDevelop. Synaptic will be used to install some needed packages and KDevelop will be the tool we will use to develop the trial application.

A.1 Installation and Configuration of OpenMP

First of all we need to install some packages which are necessary to work with this kind of development. For this purpose we use Synaptic and we install the following packages:

- binutils
- automake
- autoconf
- g++
- gcc
- m4
- libtool
- konsole
- kdevelop
APPENDIX A: INSTALLATION AND CONFIGURATION OF OPENMP AND MPI FOR C++

Figure 6: Installation and configuration of OpenMP (1).

Note that we have installed among others kdevelop and its external compiler gcc.

Since KDevelop is ready we go on creating a new program with it by:

Project menu ➔ New Project

Once project is created we need to configure it and set default by:

Project ➔ Build configuration

Figure 7: Installation and configuration of OpenMP (2).

In order to configure OpenMP we must add OpenMP linkerflag so as the compiler will recognize OpenMP:

Project menu ➔ Project Options ➔ Configure Options ➔ General
In C tab we must set GNU C compiler as C compiler and add `-pg -fopenmp` as Compiler Flag.

Figure 8: Installation and configuration of OpenMP (3).

Figure 9: Installation and configuration of OpenMP (4).
APPENDIX A: INSTALLATION AND CONFIGURATION OF OPENMP AND MPI FOR C++

Then we must choose Make Options menu and set WANT_AUTOCONF_2_5 and WANT_AUTOMAKE_1_6 variables to 0.

Figure 10: Installation and configuration of OpenMP (5).

Now it is time to write the OpenMP code. Note that it must contain omp.h library in order to get a right functionality.

Figure 11: Installation and configuration of OpenMP (6).
A.2 Installation and Configuration of MPI

Configuring MPI in KDevelop is very similar to configuring OpenMP steps. However there are some differences.

For example MPI needs some packages which we did not install in OpenMP case:

- `mpich-shmem-bin`
- `libmpich-shmem 1.0-dev`
- `libmpich-shmem 1.0gf`

Another difference is that in Configure options Compiler Command must be set as `/usr/lib/mpich-shmem/bin/mpiCC` and in this case Compiler Flags is `-O0 -G3`.  

We can see that 4 threads have been created and OpenMP is ready to work.
Moreover in Run Options menu next information must be added:

**Executable:** `/usr/lib/mpich-shmem/bin/mpirun`

**Run arguments:** `-np` executable path

**Working directory:** project path

![Figure 13: Installation and configuration of MPI (1).](image)

Of course MPI code is totally different than OpenMP code:
Figure 15: Installation and configuration of MPI (3).

Finally when we execute it we obtain the following window:

Figure 16: Installation and configuration of MPI (4).

We see that they have been created two “Hello world” messages but each of them has been executed by a different node.

A.3 Creation of a C++ profile

If we want to analyze the code execution, the delay of each method or how many times each method was called a good option is creating a profile.

First of all the program has to be compiled with pg flag activated:

```
gcc –pg –openmp –o program sourcefile.f
```
Appendix A: Installation and Configuration of OpenMP and MPI for C++

Figure 17: Creation of a C++ profile (1).

If compilation was successful a new file called *gmon.out* should be created in the source code folder.

After that we create profile as follows:

```
gprof program gmon.out > file.txt
```

Where *gmon.out* is the file generated while compilation with pg flag.

In order to see the generated file it is only needed to write next command:

```
edit fileName
```

Next windows show command execution and the result of it:

Figure 18: Creation of a C++ profile (2).
APPENDIX A: INSTALLATION AND CONFIGURATION OF OPENMP AND MPI FOR C++

Figure 19: Creation of a C++ profile (3).
CONCLUSION

Nowadays multi-core systems are more and more popular. Years ago these systems could be found in advanced computers but now they are used in a personal computer at home to a laptop of a student so everyone can work with them. Because of this, some new technologies for implementing multi-core programming were needed.

In this thesis we have seen some of the most useful technologies in this area. We have also seen some API’s such as OpenMP and MPI, some languages such as Chapel and Cilk++, Sieve compiler and Skandium Library. Actually some of them are more developed than others but all of them could be useful in certain situations.

First we reviewed OpenMP and MPI. They are the most common technologies; which are presented in this thesis. OpenMP is a very easy and simple solution for parallel programming in C. Moreover it provides a standard among a variety of shared memory architectures. On the other hand MPI is a bit more complicated than OpenMP but it does not need shared memory to work because each process is stored in a different memory space. Moreover MPI allows programmer to manage parallelization in a more customizable way.

We have presented Sieve compiler as well. It is another easier way of implementing parallelization by only introducing one marker. Despite the fact it is simple it can be used in many kinds of computers even in complex systems. Other researched
technology is Chapel. It is maybe the less useful because it needs learning a new language. In any case it is very similar to other common languages such as Java. However it is useful in some situations especially in application where parallelization is needed constantly and it will be executed exclusively in multi-core systems. The last technology we have studied is Skandium. It is less supported technology compared to others and it was developed by a Chilean university and there is not so many support or communities for it. Nevertheless it is an appealing option because it is not so much difficult to implement and its skeleton-muscle model allows programmer to configure the parallelization according to his need.

After taking an overview of each of these technologies separately, we have made a comparison between all of them in order to see their advantages and disadvantages and try to find the situations where each of them is most suitable.

Finally, in the appendix, it was shown how to install and configure in a Linux system the two best options (OpenMP and MPI) in general cases in order to begin to work with them.
2. Getting Started With OpenMP, 2998, Richard Gerber, Intel Corp.
3. Parallel Program in Fortran 95 Using OpenMP, 2002, Miguel Hermanns,
   Universidad Politecnica de Madrid, Spain.
4. Introducción a MPI (Message Passing Interface), 2006, Francisco Hidrobo,
   Universidad de Merida, Venezuela.
5. Programación de aplicaciones paralelas con MPI (Message Passing Interface),
   2007, José Miguel Alonso, Facultad de Informática UPV/EHU.
10. The Codeplay Sieve C++ Parallel Programming System, 2006, Andrew
    Richards.
11. Sieve Technology, http://offload.codeplay.com/component/content/article/102-
    sieve-technology.
13. Chapel: Cascade High-Productivity Language, 2009, Cray Inc
    Chamberlain.
15. The Chapel Language, 2010, José Gracia, HLRS.