Hybrid parallel computing applied to DNA processing

Vincent Lanore

February 2011 to May 2011

Abstract

During this internship, my role was to learn hybrid parallel programming methods and study the potential application to short sequence processing. I have designed an hybrid algorithm for Single Nucleotide Polymorphism (SNP) detection based on de Bruijn graphs and implemented it using MPI and OpenMP.

Performance tests on the laboratory's cluster show the implementation scales well up to 96 processors. I have detailed the gains of the hybrid algorithm compared to a full message-passing algorithm. These gains are lower than expected but I have proposed ways to improve both the algorithm itself and its implementation.

Contents

Chapter 1

Introduction

The internship took place in Laboratoire de Biométrie et Biologie Évolutive (LBBE) in Villeurbanne (France) under the direction of Vincent Miele, *Ingénieur de Recherche CNRS*. It lasted 14 weeks, from February 2011 to May 2011. The original French title of the internship is "Bioinformatique et calcul hybride".

Genomics is one of the main disciplines studied at the LBBE; this discipline deals with determining and studying the genome of organisms. Experiments and studies made in this laboratory range from studying a single organism to constructing trees of life involving many species. Raw data needed for those studies is obtained through a process called *sequencing* which extracts data from cells to obtain strings. Until recently, sequencing was an expensive and slow process. Nowadays, the development of new technologies called Next Generation Sequencing (NGS) techniques made sequencing quicker and affordable for many laboratories. It is an important step forward for genomics since it allows many new possibilities of studies. A single NGS machine can produce hundreds of gigabytes of data per day. Such an amount of data requires computers to be processed and, even so, is so large that it presents a challenge. Thus, genomics offers many challenging problems, notably in string processing algorithmics.

Besides the field of parallel computing is evolving consequently to the slowing down of the Moore's law and the multiplication of massively multi-core architectures. Nowadays in bioinformatics, laboratories often have large processing clusters that, in France, usually consist in computers with tens of processors and large amounts of memory. There is such a cluster in LBBE. Classical parallel programming techniques may not make the most of such architectures. Hybrid parallel programming is a parallel programming method that is supposed to suitable for architectures with many cores per machine. Hybrid parallel programming is quite new and has never been widely used even in pure computer science applications. As such, it has never been studied at the LBBE's bioinformatics department although the laboratory's cluster seems perfect for it.

My goal during the internship was to learn hybrid parallel programming and to study its potential applications to bioinformatics through the problem of Single Nucleotid Polymorphism detection (SNP detection). I was then advised to study theoretical and practical aspects of the problem and to implement an hybrid SNP detection program to be run on the laboratory's cluster. In the present report I will first present my study of hybrid parallel programming. Then I will explain the SNP detection problem and give a few useful elements of theory. Finally I will present `HyBu`, the hybrid parallel algorithm I conceived and implemented.

Chapter 2

Hybrid parallel programming

This part presents hybrid parallel programming and the two APIs that will be used for the implementation : OpenMP and MPI. Since hybrid parallel programming was completely new in LBBE, I worked on this part in almost complete autonomy. This part is mostly a synthesis of courses and tutorials about hybrid programming, MPI and OpenMP.

2.1 Concept and motivation

Parallel programming on CPUs relies on the use of several processes or threads to run instructions in parallel. Information transfer between processes can be done either by sending messages between processes or by having shared memory where one process can leave data to be used by other processes.

Message-passing works for all applications since messages can be sent between processes on the same computer or over networks. Compared to accessing memory, message-passing is costly (the added cost is called *overhead*) and requires the processes to handle both sending and receiving. Shared memory on a single computer is very fast since accessing the shared memory is basically a simple memory access. However, it requires an architecture that allows shared memory access.

The idea behind hybrid parallel programming is to take the best of both approaches with programs that use shared memory where possible and message-passing to communicate between shared-memory nodes. Such programs avoid overhead produced by message-passing while being compatible with any CPU-based architecture. However, hybrid programs are harder to design since work balancing must be made on two levels (between nodes and among threads within a node) and since message-passing and shared memory technologies were not designed to work together.

Although the required technology has existed for some time, hybrid parallel programming has begun developing only recently along with the multiplication of Symmetric Multiprocessing clusters (SMP clusters) i.e clusters of shared memory multi-processor computers. Hybrid computing techniques are continuously improved and it is difficult to state which problems will benefit from hybrid computing.

2.2 Technical aspects

Approaches to message-passing and shared memory can vary greatly from one specification to another. Within the context of the internship, I was strongly advised to use OpenMP for the shared memory part and MPI for the message passing.

2.2.1 MPI presentation and basics

MPI is a message-passing library interface specification. It is widely used, particularly in cluster computing.

A MPI program is meant to be run on several processors. Each running copy of the program is called a *process* and distinguishes from other processes using its unique *process number*. MPI deals with message passing which means it introduces functions to *send* and *receive* messages between processes. A message can be sent from one process to another : it is called a *point-to-point communication*. There are also functions which imply communication between all the processes ; they are called *collective communications*. A communication can be *blocking* which means the function called will only exit when the communication is over or *non-blocking* which means the function initiating communication will return immediately but that the completion of the communication will need to be checked afterwards [5].

MPI has many other features [11] that will not be detailed here.

2.2.2 OpenMP presentation and basics

OpenMP is an API for parallel computing on shared memory architectures. An OpenMP program is a sequential program with added *directives* that apply to *subsections* of code and specify how parallelization is done. OpenMP is based on the *fork-join model* : at runtime, the program executes sequentially until a *parallel subsection* is met at which point *threads* are created (*fork*) and execute the code in the subsection; when it is done, the program destroys the threads (*join*) and continue executing sequentially until the next parallel subsection [1, 6].

OpenMP allows all the threads to access the same places in memory but does not provide *thread-safe* data structures i.e data structures that can be accessed correctly in parallel by several threads. Instead, OpenMP provides *locks* and *critical subsections*. Locks are meant to be locked and unlocked by threads. A lock is designed so that it is impossible for several threads to lock the same lock. If we force threads to lock a lock before accessing a data structure and to unlock it afterwards then only one thread can access the data structure at a time. Critical subsections are simply subsections of code that can be executed by only one thread at a time.

Many more OpenMP functionalities exist [13] but we wont detail them here.

Although it is popular, OpenMP was not an obvious choice since there are other commonly used APIs for shared memory parallelism such as Intel TBB or even POSIX threads. POSIX threads were rejected since it was too low-level for our needs. TBB has high-level functionalities but we preferred OpenMP which was more established and widely used.

2.2.3 Hybrid computing

An hybrid MPI/OpenMP program is a MPI program which uses OpenMP directives. It is run like an ordinary MPI program. In order to make use of the full potential of the architecture we need each thread to be able to send and receive MPI messages which means we need a *thread-safe MPI implementation* [9] [4].

Given an architecture with n shared memory nodes each with m processors, an hybrid program would ideally be run with one process per node each spawning m threads for a total of $m \times n$ threads distributed among the $m \times n$ processors. This way we maximize the use of shared memory over message-passing. Practically, we will see that we may need to have l processes per node and m/l threads per process (see 4.4.3).

Chapter 3

The SNP detection problem

This part presents the bioinformatics problem to which hybrid programming will be applied and presents De Bruijn graph, a theoretical structure that is useful in solving DNA processing problems [14]. Compared to the preceding part about hybrid programming, I was much more guided for this part which is mostly a synthesis of articles I was advised to read.

3.1 Preliminary notions

DNA stands for Deoxyribonucleic Acid. It is composed of two long polymers (the strands) of units called nucleotides arranged to form a double helix structure. The nucleotides encountered in DNA strands can be of four types: Adenine (A), Cytosine (C), Guanine (G) and Thymine (T). Each nucleotide on one strand is linked to a nucleotide on the other strand according to the following pattern: A is always linked to T (and conversely) and G is always linked to C (and conversely). The linked pair of nucleotides so-formed is called a *basepair*. The sequence of nucleotides of one strand can be deduced from the sequence of the other strand using this linking rule. The resulting sequence is called the *reverse complement* of the original sequence. All the DNA molecules present in a cell of an individual form its *genome*. Genome is made of *chromosomes*, each chromosome containing a single DNA molecule. Depending on the species, chromosomes may be paired. A *Single Nucleotide Polymorphism* (SNP) is a DNA sequence variation between individuals of a same species or between two paired chromosomes. It consists in a difference of one nucleotide between the two DNA sequences. SNPs are the most common DNA sequence variation.

Sequencing is the process that takes a DNA molecule and gives its sequence of nucleotides. Today sequencing methods do not allow the sequencing of whole DNA molecules. Instead, molecules are broken into smaller pieces which are sequenced. The resulting sequences are called *reads*. The output of sequencing is a list of reads that need to be assembled to get the complete sequence (*assembly*). In order to ensure there is enough information to assemble the whole sequence afterwards DNA is not sequenced from a single molecule but from several identical ones. This means that a section of DNA may be covered by several reads. The ratio between the sum of the length of the reads and the length of the DNA to be sequenced is called the *coverage*. Reads may be subject to *sequencing errors*: a nucleotide may be misread resulting in a wrong nucleotide in one read.

Remark In the following, any DNA sequence will be considered strictly equivalent to its reverse complement since they carry the same information.

3.2 Background

Whole-genome sequencing was not possible until the 1970s with the development of rapid sequencing technologies such as the Sanger method. Such methods were slow and expensive. A second important breakthrough happened in the late 2000s with the rise of Next-Generation Sequencing (NGS) technologies such as Roche 454, SOLiD Applied Biosystems or Illumina Solexa. These methods dramatically reduced the overall cost and improved the throughput of sequencing. Reads tend to be short (25 to 500 bp) and inaccurate (depends on the method). Coverage obtained with NGS methods ranges usually from $2\times$ to $100\times$.

Algorithms that existed to process and assemble Sanger reads were unadapted to NGS data. It led to new developments in short read processing programs. Pevzner et al. [14] proposed an assembly method based on a De Bruijn graph representation that is widely used today in assembly programs like [10, 15, 16, 2].

3.3 De Bruijn graphs

3.3.1 Definition

Definition 1. *A DNA sequence is a string on alphabet $\mathcal{A} = \{A, T, G, C\}$. A base is a character from $\mathcal{A}$.*

Definition 2. *The set of k-mers $\mathcal{K}$ of a set of reads $\mathcal{R}$ is defined by: $u \in \mathcal{K}$ if and only if $\exists r \in \mathcal{R}$ such that u is a substring of r of length k.*

Remark Reads and k-mers are DNA sequences.

Definition 3. *For a given k. The De Bruijn graph $\mathcal{G}$ of a set of reads $\mathcal{R}$ is a graph (V, E) such that E is the set of k-mers of $\mathcal{R}$ and such that $(u, v) \in V$ if and only if u and v overlap by $k-1$ bases.*

De Bruijn graphs were introduced because assembling the original sequence is done by finding an eulerian path in the graph where former approaches (overlap graphs) required a hamiltonian path which is a NP-complete problem [14].

Notation If u is a k-mer then $\forall i \in \{0, \dots k-1\}$, u_i is the i^{th} base of k-mer u.

3.3.2 SNPs in De Bruijn graphs

Definition 4. *A k-mer $u = u_0u_1 \dots u_{k-1}$ is a right neighbor (resp. left neighbor) of k-mer $v = v_0v_1 \dots v_{k-1}$ (where $\forall i \in \{0, \dots k-1\}, u_i \in \mathcal{A}$ and $v_i \in \mathcal{A}$) if and only if $\forall i \in \{1, 2, \dots k-1\}, u_i = v_{i-1}$ (resp. $v_i = u_{i-1}$).*

Definition 5. *A right (resp. left) branching point in a De Bruijn graph is a k-mer which has at least 2 right (resp. left) neighbors.*

Considering we move from left to right (moving from one k-mer to its right neighbor) along a linear subsection of the De Bruijn graph of a set of reads. Each k-mer will have only one right neighbor as long as we encounter no DNA sequence variation or random repeat of a k-mer. When encountering a SNP, one k-mer u will have two right neighbors $u_1u_2 \dots u_{k-1}a$ and $u_1u_2 \dots u_{k-1}b$ (where $\{a, b\} \in \mathcal{A}^2$). Then we can continue on both branches during k steps (the i^{th} k-mers encountered will have the form $u_i \dots u_{k-1}av_0 \dots v_{i-1}$ and $u_i \dots u_{k-1}bv_0 \dots v_{i-1}$ respectively for each branch) before we meet a k-mer $v = v_0v_1 \dots v_{k-1}$ that joins the two branches. The resulting structure is called a *mouth* or a *bubble* [8] and is typical of SNPs in De Bruijn graphs (see figure 3.1).

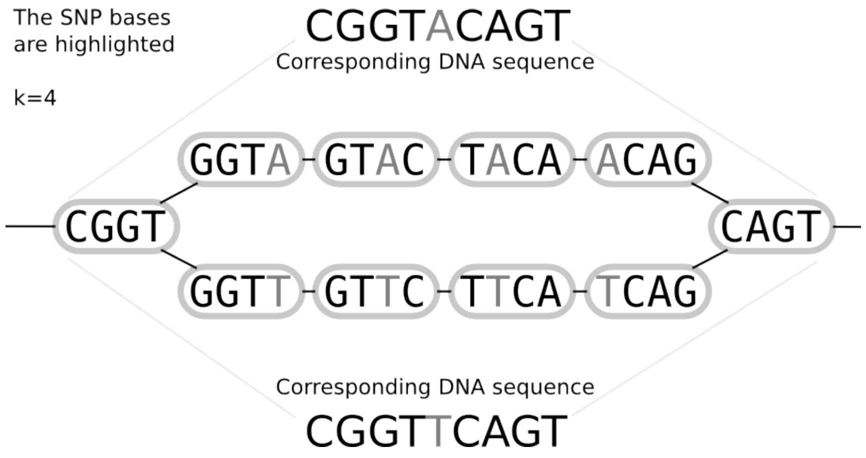

The SNP bases are highlighted
k=4
CGGTACAGT
Corresponding DNA sequence
GGTA
GTAC
TACA
ACAG
CGGT
CAGT
GGTT
GTTC
TTCA
TCAG
Corresponding DNA sequence
CGGTTCAGT

Figure 3.1: A SNP bubble in a De Bruijn graph.

Chapter 4

HyBu

This part presents `HyBu`, the parallel algorithm I designed and implemented to address the SNP detection problem. It is strongly inspired from existing algorithms like ABySS but most of it is my own work like the adaptation to hybrid programming or the parallelization of the SNP detection algorithm.

4.1 Distributed De Bruijn graphs

All the programs based on De Bruijn graphs mentioned above are single-threaded applications which have a hard time dealing with large datasets such as mammalian genomes. Rare parallel applications have been developed to tackle this problem. They include notably the ABySS assembler [7]. Such programs can be run on large clusters to allow processing large datasets within a reasonable amount of time for even the largest genomes.

ABySS is a parallel DNA assembler from short reads that uses a distributed representation of the De Bruijn graph. This representation is based on a hash function that is used o distribute the k-mers among processors. Each process stores only the information for the k-mers it owns and knows where each k-mer is stored according to the hash function. The first step in the ABySS algorithm is to extract each k-mer from the reads and to send it to the process which owns it. I was advised to use this distributed representation of the De Bruijn graph for my own SNP-detection algorithm. It needed to be adapted to hybrid parallel programming but was a good starting point.

4.2 Algorithm

The algorithm is divided into three stages (numbered 1, 2 and 3). First, the distributed representation of the De Bruijn graph is built by distributing k-mers among the MPI nodes. Then, adjacency information for each k-mer is computed [7]. Finally, the algorithm searches SNPs bubbles in the built De Bruijn graph.

Since it is an hybrid parallel algorithm the work will be divided among processes and each process will then divide the work among several threads. Each process has its own data in shared memory that is accessible by all its threads; we call this data the *local* data of the process. Notably, each process will have a hashtable indexed by k-mers [7] that contains information about them.

Pseudo-code conventions For the sake of simplicity we will present hybrid algorithms in two separate parts. The reception of messages will be handled separately from the core of the algorithm by another thread (for a total of $2 \times m \times n$ threads for $m \times n$ processors). Though it may not reflect implementation (see 4.4.2) it is much easier to present algorithms that way.

Note also that there will not be any "for all processes" or "for all threads" and that it is implicit that the pseudo-code is meant to be run on $m \times n$ processors organized in processes and threads as described above (see 2.2.3).

4.2.1 K-mer distribution

The algorithm relies on a representation of the De Bruijn graph distributed among processes. For each process there are threads that work concurrently on the local k-mers in shared memory. The hash function used is the same as in [7] (i.e a generic hash function is computed on the k-mer and its reverse complement and the results are combined using a commutative operation like XOR) except it is computed only on the $k-2$ middle bases of the kmer (i.e if $u ==_0 u_1 \dots u_{k-1}$ is a k-mer, the hash function will be computed on $u_1 \dots u_{k-2}$). This new hash function will be useful for the second part of the algorithm.

Algorithm 1 K-mer distribution

```
for all read $r$ in input file do
  for all kmer $u$ in $r$ do
    send $u$ to hash($u$)
  end for
end for
for all received kmer $u$ do
  store $u$ in local hashtable
end for
```

The input reads are uniformly split among the processes and then uniformly split among the threads. This distribution concerns the input reads and should not be confused with the distribution of k-mers with the hash function. Each process has a hashtable designed to contain k-mers that is accessible (for reading and writing) concurrently by several threads.

Each thread goes through its reads and divides them into k-mers. For each k-mer, the process it belongs to is computed using the hash function and a message containing the k-mer is sent to this process. Upon reception, each k-mer is added to the hashtable of the process.

4.2.2 Building the De Bruijn graph

Now that each process has its k-mers it is time to compute *adjacency information* for all of them [7]. Adjacency information consists in knowing the neighbors of a given k-mer. Practically, as a k-mer and its neighbors overlap on $k-1$ bases, a k-mer can only have up to 8 neighbors (4 on the left for each possible base, 4 on the right for each possible base).

Let us consider a k-mer $u = u_0u_1 \dots u_{k-1}$. Its left neighbors will be $Au_0u_1 \dots u_{k-2}$, $Tu_0u_1 \dots u_{k-2}$, $Gu_0u_1 \dots u_{k-2}$ and $Cu_0u_1 \dots u_{k-2}$; its right neighbors will be $u_1u_2 \dots u_{k-1}A$, $u_1u_2 \dots u_{k-1}T$, $u_1u_2 \dots u_{k-1}G$ and $u_1u_2 \dots u_{k-1}C$. Since the hash function described above is computed only on the $k-2$ middle bases it will give the same result for all the four left neighbors of u (computed on $u_0 \dots u_{k-3}$) and the same results for all the right neighbors (computed on $u_2 \dots u_{k-1}$). It means that all neighbors on a given side (left or right) of a given k-mer will belong to the same process.

The k-mers stored in the processes' hashtables are uniformly split among the threads. Each thread goes through its k-mers and, for each k-mer u, computes the hash function for hypothetical left neighbors (which gives one process number) and right neighbors (which gives another process number). Then it sends a message containing u and a side label to each of those two processes.

Algorithm 2 Building the De Bruijn graph

```
for all kmer $u$ in local hashtable do
  for all right neighbor $r$ of $u$ do
    send $u$ to hash($r$) with label RIGHT
  end for
  for all left neighbor $l$ of $u$ do
    send $u$ to hash($l$) with label LEFT
  end for
end for
```

```
for all received kmer $u$ do
  if label is RIGHT then
    for all four possible right neighbors $n$ of $u$ do
      if $n$ is in local hashtable then
        add $u$ to left neighbors of $n$
        if $n$ already has a left neighbor then
          add $n$ to list of left branching points
        end if
      end if
    end for
  else
    //Same for RIGHT label
  end if
end for
```

The side label is LEFT if the message is sent to the owner of potential left neighbors of u, RIGHT otherwise.

When a thread receives a k-mer u and a side label, it computes the 4 potential left (if the side label is LEFT) or right (otherwise) neighbors and looks for them in the hashtable. If neighbors are found their adjacency information is updated to add u to their known neighbors. For example: if $u = u_0u_1 \dots u_{k-1}$ is received with label LEFT and if $v = xu_0u_1 \dots u_{k-2}$ (where $x \in \mathcal{A}$) is in the local hashtable then v's adjacency information will be updated with "the right neighbor ending with x exists". When adding a neighbor to adjacency information, if the k-mer already has one neighbor on the same side then the k-mer is a (left or right) branching point. Found branching points are added to a list in the process' shared memory with their orientation (left or right).

4.2.3 Finding the SNPs

Now that we have computed adjacency information and listed branching points we can look for SNP bubbles. The idea is to select a branching point and to move along the branches until they join ; if both branches are of size $k+1$ then they form a SNP bubble. This walk on the De Bruijn graph will be made with *cursors*. A cursor is a token meant to be passed from process to process. It contains the following data: the branching point s from which the cursor originated, its target t which is the k-mer the cursor is trying to reach and an integer counter c which indicates the number of steps the cursor has made so far. *Sending a cursor to a k-mer u* means setting t to u, incrementing c and sending a message with s, t and c to the owner of t.

Each process has its own left branching point list which it splits uniformly among its threads. For each left branching point, the thread creates and sends a new cursor to each of its right neighbors. Upon reception of a cursor the receiving thread looks at the target :

- if it is a dead-end the cursor is discarded ;

Algorithm 3 Finding the SNPs

```
for all kmer $u$ in list of left branching points do
  create cursor $C$ from $u$
  for all left neighbor $l$ of $u$ do
    send $C$ to $l$
  end for
end for
for all received cursor $C = (s, t, c)$ do
  if $t$ is a right-branching point then
    if $C$ found in stored right branching points then
      print the two sequences showing the SNP
    else
      store $C$
    end if
  else if $n$ has left neighbors and $c < k + 1$ then
    send $C$ to each of them
  end if
end for
```

- if it is a right branching point then the cursor is stored until other cursors arrive to the same branching point ;
- if it is a right branching point and other cursors have already arrived to it then the arriving cursor is compared to them ; if one of them has the same S and both c counters equal $k + 1$ (cursors are identical) then a SNP has been found ;

then the cursor is sent to the right neighbors of its target t unless its counter c has reached $k + 1$ in which case it is discarded.

When a SNP is found, the sequence surrounding the SNP can be easily extracted from the cursors' data. Indeed, s (a left branching point) and t upon arrival (a right branching point) are the two closest k-mers on the De Bruijn graph unaffected by the SNP. The only missing data is the different possible values for the SNP base but it can be found in t's adjacency information (or s's but t is locally accessible by the thread who receives the cursor). If $u = t_1 t_2 \ldots t_{k-1} a$ and $v = t_1 t_2 \ldots t_{k-1} b$ (where $\{a, b\} \in \mathcal{A}^2$) are the two right neighbors of $t = t_0 t_1 \ldots t_{k-1}$ and if $s = s_0 s_1 \ldots s_{k-1}$ then $s_0 s_1 \ldots s_{k-1} a t_0 t_1 \ldots t_{k-1}$ and $s_0 s_1 \ldots s_{k-1} b t_0 t_1 \ldots t_{k-1}$ are DNA sequences of length $2k + 1$ showing the SNP.

4.3 Analysis

4.3.1 Complexity

Property 1. *Let $\mathcal{R}$ be a set of reads, d the number of distinct k-mers in the set of k-mers of $\mathcal{R}$ and c the coverage. Let n be the number of processes, m the number of threads per process and $p = m \times n$ the total number of processors. If accessing the hashtable and message sending are in $O(1)$ and allow full parallel use then the cost of the first part of the algorithm is $O(\frac{c \times d}{p})$; the cost of the second part is $O(\frac{d}{p})$.*

Sketch of proof The first part sends one message and do one attempt of writing into the hashtable per k-mer directly extracted from the reads (before removing multiplicity) ; by definition of c there are $c \times d$ such k-mers. The second part sends 2 messages and do a constant number of accesses to the local hashtable for each k-mer previously stored on each process' hashtable. Since each k-mer is stored in exactly one hashtable, there are d such k-mers.

Remark Complexity of part 3 is difficult to estimate since it depends on the number of branching points and would require a stronger hypothesis regarding the shape of the graph. Thus, complexity of part 3 will not be detailed here.

4.3.2 Gains from hybrid approach

The purpose of hybrid parallel programming is to maximize thread communication via shared memory at the expense of message-passing. With that in mind we want to know how many messages have been avoided compared to a full message-passing approach.

Property 2. *Let n be the number of processes, m the number of threads per process and $p = m \times n$ the total number of processors. Let u be a k-mer and $i \in \{0, \dots p-1\}$ a process number. In a full message-passing approach with p processes, the probability that i owns k-mer u is $\frac{1}{p}$. Let $j \in \{0, \dots n-1\}$ be a process number. The probability that j owns u in a hybrid approach is $\frac{1}{n}$.*

It means that the hybrid approach avoids only $\frac{1}{n}$ of messages. If $n > 2$ the gain is low. This comes from the fact that we use a generic hash function to distribute k-mers which means the location of a k-mer can be considered random. Thus there is no property of the k-mer distribution we can use to improve the number of messages avoided. Finding a hash function that ensures such a property is difficult since we know very little about the shape of the De Bruijn graph before it is actually built.

4.3.3 A potential improvement

Let $\mathcal{R}$ be a set of reads, d the number of distinct k-mers in the set of k-mers of $\mathcal{R}$ and c the coverage. The first part of the algorithm sends $c \times d$ messages which means a same k-mer can be sent several times (c times on average). We may want to prevent a same process from sending a same k-mer several times. This means removing multiplicity locally before the first part of the algorithm.

In a full message-passing approach with p processors and coverage c there will be $\frac{c}{p}$ identical k-mers on average per process. Since coverage is usually lower than p (coverage vary between $2\times$ and $100\times$ while targeted architectures have tens of processors) this will result in very low message avoidance.

Conversely, an an hybrid approach the number of MPI processes is much lower. Let n be the number of processes, m the number of threads per process and $p = m \times n$ the total number of processors. There will be $\frac{c}{n}$ identical k-mers on average per process. Since n is usually very low (often < 8) and since c can be as high as 100 with today sequencing techniques, the hybrid approach can result in a massive gain for the first part of the algorithm.

4.4 Implementation

As discussed above, implementation was made in *C++* with *openMP* for the shared-memory part and with *MPI* for the message-passing part. Libraries needed to be non-proprietary and cross-platform. *Open MPI* was the MPI implementation used; *g++* was the openmp-compatible C++ compiler used.

4.4.1 Constraints and goals

The targeted architecture is the LBBE's cluster which is composed of 48-core machines with 240Gb of memory. The primary goal of the algorithm and its implementation is *scalability.* An algorithm is said *to scale* or to have a perfect scalability if the execution time is inversely proportional to the number of processors used for its execution. Since it is practically very difficult to have a perfectly-scaling implementation, we say that the implementation scales if the practical execution times are close to the theoretical perfect-scaling execution times.

4.4.2 Implementation problems

Quality of MPI and OpenMP implementations. MPI and OpenMP implementations were not designed to work together. Notably, finding a thread-safe implementation of MPI can be a problem since most widely-used open-source MPI implementations experience problems with this functionality. Moreover, OpenMP implementation targets mostly classical multicore processors with no more than 8 or so cores. It means that scalability problems begin to appear above 8 threads per process. Avoiding the use of functions that cannot be fully parallelized such as memory allocation functions can push back this limitation. Even so, scalability with tens of threads per process is very difficult to obtain [9, 4].

Balancing load and receiving messages. An easy way to send and receive messages (and probably the most intuitive way to implement the algorithm above) is to use separate threads for sending and receiving, but if we do so, we loose control over the scheduling which could result in poor execution time balancing and performance loss.

Moreover we cannot use blocking communications since the algorithm (especially the SNP-finding part) does not let us know from which process the next message will come.

This results in the use of non-blocking communications with reception and sending taking place in the same thread. It has several advantages: the latency of the network can be hidden with extra computation, the low level of synchronization improves performance and the termination of the algorithm is easier.

Arrays. We want to be able to access data in parallel with different threads but most of commonly available data structure implementations are not thread-safe. Using locks is a solution but it threatens scalability. The easiest way to proceed is to use arrays since they can be written and read in parallel by threads which work on disjointed segments. Moreover MPI uses arrays to send and receive data.

This resulted in an extensive use of arrays in the program. It lead to many memory allocation problems and made the code difficult to read.

False sharing. When several threads work on the same array and write cells which are close in memory it may happen that these two cells have been loaded in the same cache line. OpenMP prevents the threads from accessing the cache line at the same time. As one thread may have to wait before accessing the cache line this is no longer a real concurrent access. It may result in scalability problems. This phenomenon is called *false sharing* [4].

Since the program uses many arrays to store thread-related information false sharing was a problem for the early scalability tests. The solution was to re-design memory allocation so that data used by different threads is adjacent in memory.

Hashtable implementation. Commonly available hashtable implementations are not thread-safe. Since hashtables can be resized when adding an element, even careful locking of buckets will fail to provide safe parallel access. Very few thread-safe hashtable implementations exist and I have not found one that could easily be used in the code. Implementing one is possible ([3], for example) but is beyond the scope of the internship.

The chosen solution was to use several hashtables with a custom hash function to distribute keys among them. Each hashtable can be safely locked by a thread which wants to write in it. Once wrapped correctly these hashtables are equivalent to one bigger hashtable which is slower (since the custom hash function is not optimized) but allows some safe concurrent access.

4.4.3 Performance tests

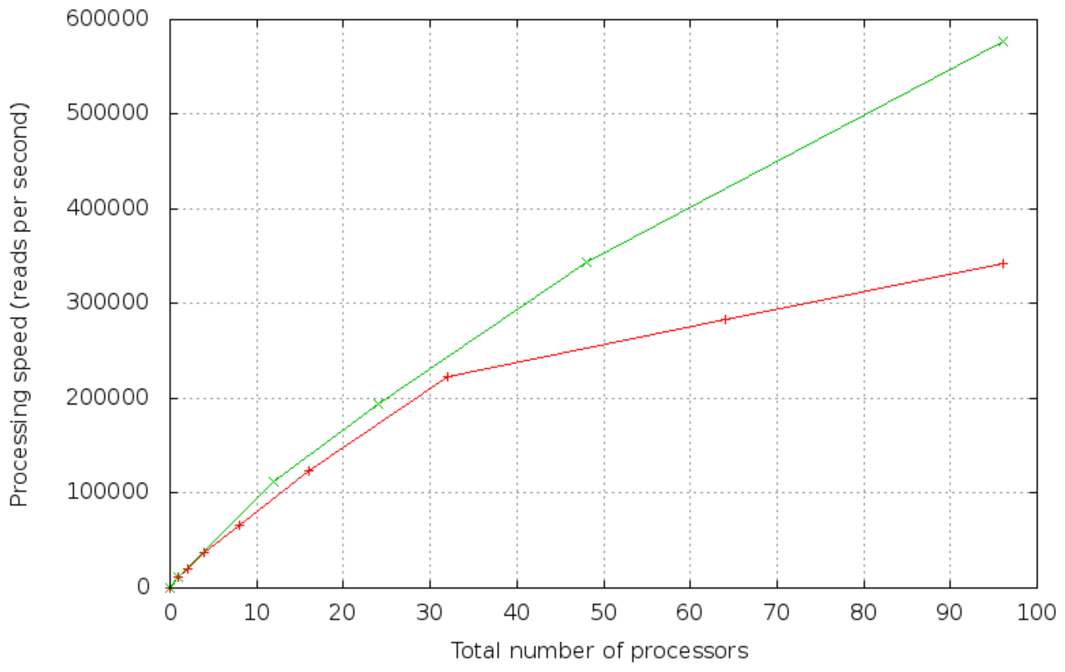
Performance tests allow us to measure the scalability of the implementation of the algorithm. Performance tests were made on two 48-core machines (insert technical specs). We used the code of the first two parts of the algorithm (building the distributed De Bruijn graph) for the tests since implementation of part 3 of the algorithm was not fully functional at the time. Tests were made with $k = 16$. Tests were made on two datasets :

- *Dataset1* is a simulated dataset with 5 million reads of size approximatively 35 extracted from a randomly generated DNA sequence of size 2.5 million.
- *Ecoli* is a real dataset [12]. The reads come from the genome of a bacteria named *Escherichia coli*. The dataset contains about 7 million reads of size 35 while the whole *Escherichia coli* genome has size 5 million.

Tests were made for various numbers of processes and threads per process. Figure 4.1.A) shows performance with one process per machine and with 6 processes per machine. For a given number of processors p there are 6 times more threads per process on the lower curve. For low values of p the two curves are close which means equal performances. Starting from $p = 32$ performance of the algorithm with one process per node is remarkably lower. This means scalability is poor when the number of threads per process is greater than 16 while scalability is good with several processes per node. This illustrates the fact that it is difficult to have OpenMP programs that scale above 8 threads (see 4.4.2).

Performance with *ecoli* (Figure 4.1.B)) presents an almost linear increase in processing speed with the increase of the number of processors. Processing speed with 12 threads per process is 68% of 12 times the processing speed with 1 thread per process. Performance with four 12-core processors is significantly lower than with 48 processors and could explain the difference with theoretical expectations. Although the algorithm scales well with *ecoli*, it is slower than with *dataset1*. This can be explained by the fact that coverage is lower which means there are more k-mers in average per read. Moreover the algorithm's output showed that there are much more branching points than with *test* which is due to the fact that real DNA is prone to sequence repetition.

A)



B)

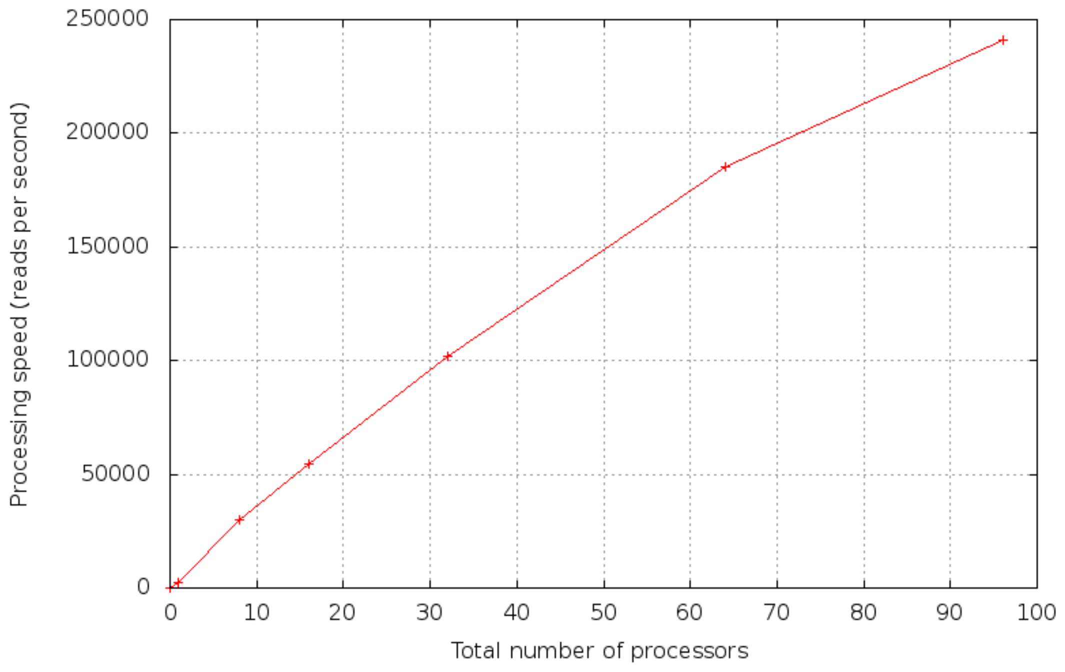


Figure 4.1: Performances tests of the algorithm on two nodes of the LBBE's cluster. Each node is made of four 12-core quadri-AMD opteron 2200MHz with 64 gigabytes of memory for a total of 48 processors and 256 gigabytes of memory. A) Dataset *dataset1* with 6 processes per node (upper curve) or 1 process per node (lower curve). B) Dataset *ecoli* with 4 processes per node.

Chapter 5

Conclusions

In the course of this internship I learned hybrid parallel programming and applied it to the SNP detection problem. I proposed an algorithm called `HyBu` inspired from existing work on De Bruijn graphs that solves the problem and I implemented it using OpenMP and MPI.

It turned out that the SNP problem and the De Bruijn graph approach do not allow high gains from the hybrid parallel programming approach. It also turned out hybrid parallelism is not easy to implement. There is still room for improvement. Removing k-mer multiplicity in parallel on each process before running the algorithm would greatly improve performance on datasets with high coverage and it could not be done with a full message-passing algorithm. Implementation could also be improved; I suggested that the scalability of an algorithm with OpenMP depends highly on how much some thread-safe functions are used. Optimizing the implementation so as to use less these functions, could make the algorithm scale up to more threads per process.

However, I managed to get most of the implementation to work and I tested it on the LBBE's cluster. These tests showed which parameters to use to get the better scalability and allowed to draw some practical conclusions about the algorithm. I obtained a good scalability up to 96 processors. Since most DNA processing algorithms are sequential, having implemented a parallel algorithm that scales with that many cores can be considered an achievement in itself. Although the conclusions regarding the suitability of hybrid parallel programming applied to algorithms based on De Bruijn graphs may not be as positive as expected, the internship laid the foundations for further study of hybrid parallel programming in LBBE.

Bibliography

[1] Blaise Barney. *OpenMP*. Lawrence Livermore National Laboratory, 2011. https://computing.llnl.gov/tutorials/openMP/.

[2] Patrick S Schnable Benjamin G Jackson and Srinivas Aluru. Parallel short sequence assembly of transcriptomes. *BMC Bioinformatics*, 2009.

[3] Groote J.F. Gao H. and Hesselink W.H. Almost wait-free resizable hashtables. *18th International Parallel and Distributed Processing Symposium*, 2004.

[4] HPC@LR. Formation openmp avancé, mar 2011. http://www.hpc-lr.univ-montp2.fr/node/127.

[5] IDRIS. *Cours MPI*, 2011. http://www.idris.fr/data/cours/parallel/mpi/choix_doc.html.

[6] IDRIS. *Cours OpenMP*, 2011. http://www.idris.fr/data/cours/parallel/openmp/.

[7] Shaun D. Jackman Jared T. Simpson, Kim Wong. Abyss : A parallel assembler for short red sequence data. *Genome Research*, 2009.

[8] Pierre Peterlongo; Nicolas Schnel; Nadia Pisanti; Marie-France Sagot; Vincent Lacroix. Identifying snps without a reference genome by comparing raw reads. *String Processing and Information Retrieval*, 2010.

[9] Maison De La Simulation. *Programmation hybride MPI/OpenMP*, oct 2010. http://www.maisondelasimulation.fr/Phocea/Vie_des_labos/Seminaires/index.php?y=2010&id=8.

[10] Guillaume Marçais1 and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 2011.

[11] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard version 2.2*, sep 2009.

[12] NCBI. Sequence read archive. http://www.ncbi.nlm.nih.gov/sra/SRX012986?report=full.

[13] OpenMP Architecture Review Board. *OpenMP Application Program Interface version 3.0*, may 2008.

[14] Haixu Tang Pavel A. Pevzner and Michael S. Waterman. An eulerian path approach to dna fragment assembly. *PNAS*, 2001.

[15] Douglas L. Maskell Yongchao Liu, Bertil Schmidt. Decgpu: distributed error correction on massively parallel graphics processing units using cuda and mpi. *BMC Bioinformatics*, 2011.

[16] Daniel R. Zerbino and Ewan Birney. Velvet: Algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, 2008.