Abstract

This project has implemented a number of algorithms into parallel on the XMOS device. These implementations will be scaled on the parallel device, and the scalability and the efficiency of these implementations will be investigated. These implementations will not only demonstrate the power of parallelization, but also show how parallelization has been pulled back by a set of hindering forces that go against parallelization. By performing an in-depth analysis on these implementations, we have detected a number of these hindering forces. At the end of the project, we optimized and re-implemented some of these implementations, and tried to find some guidelines about how to overcome or reduce the negative effects of these hindering forces.

This project also shows how we can make a compromise between efficiency and flexibility during the parallel implementations. It demonstrates how an implementation gets more flexibility but loses efficiency. It will also demonstrate how an implementation optimized on a particular device can obtain more efficiency, but it then becomes unable to fit into other devices.

Those implementations will be executed on a 4-core and a 64-core XMOS device. The XMOS devices are based on distributed memory parallel architecture and use a message passing programming model. It can have up to 512 threads working simultaneously. That is relatively a very large number for such a portable device, and it is sufficient for us to scale our algorithms on it.

I implemented the Merge Sort and the N-Queens Problem into many different implementations on the XMOS device.
I demonstrated the power of parallelization, how can the parallelization being pull back and how an algorithm with a frequent data communication unable to fit the parallelization which are all shown in chapter 4.
I deeply analysed the implementations in chapter 4.3 and chapter 4.5 and detected a set of hinder forces which go against parallelization.
I designed a ring connection architecture for core connections in chapter 3.3.6, which has a great scalability and demonstrated how to make compromise between flexibility and efficiency together with the parallel Merge Sort implementation.
I deeply optimized the Merge Sort implementation on the 4 core XMOS device and obtained a higher speed up while using less number of cores than the original implementation. The design is shown in chapter 3.3.3 and the result is shown in chapter 4.6.

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Wen Luo, September 2011

Contents

Chapter 1: Introduction

Parallel computation has become an urgent application as the clock rate for a single processer has already reached the limit as described in chapter 2.1. In the meantime, the hardware has already taken the first step to parallelization. Dual-core processors have become a basic for mobile phones, tablets and personal computers. Some high-end computers and servers even have a 4- or 8-core processor. [1] However, there is only a small number of software that can take full advantage of the multicore hardware.

As the core numbers for a single processor may have an exponential increase in the near future, will the overall performance increase with the growth of core numbers? Can the software engineer easily transfer their code to fit various core numbers well? Both the hardware and the software engineers have been asked these two questions on our path to parallelization.

By going through this project, you will see there are a number of hindering forces against parallelization. These hindering forces are typically some factors that make the parallel computation unable to make full use of the parallel processor. Some of the factors can be avoided but others are inevitable. This project will try to identity them and try to find some general solutions for dealing with these factors. Software engineers writing parallel programs need to be aware of these factors and try their best to avoid or reduce the drawbacks caused by these hindering forces.

And because of these hindering forces, some algorithms may hardly speed up when going more parallel, and some will even slow down. Having too many cores is a waste, and we must make sure we really need such a huge number of cores before we scale up the processors. This project will show how some particular kinds of algorithms speed up when running them on different number of cores.

In the field of parallel computation, efficiency is not everything. We need to make a compromise between efficiency and flexibility. A parallelized code can be written in many different ways based on similar algorithms as the data flow can be various. In general, the more efficiency being generated, the less flexible the code will be. A code can be deeply optimized for a particular core number to generate more efficiency, but the code will lack flexibility. It would be a disaster for the software engineers if they have to rewrite the whole code every time the core number increased. This project will introduce several ways to write a parallel program, and it will try to find out where the balance point is between flexibility and efficiency.

All the codes used in this project were tested on the XMOS XC-1A Development Kit [7] or the XK-XMP-64 Development Board [16]. The XMOS processors are based on the Message Passing architecture, which can be easily scaled up. The XC-1A Development Kit has a single XS1-G4 [7] processor, which has 4 cores in a single chip and can have a maximum of 8 threads running on each core. The XK-XMP-64 Development Board is scaled up by 16 XS1-G4 processors. This means it has 64 cores on the development board, and it can have a maximum 512 threads in total running on the board.

1.1 Aims and Objectives

This project will do some parallel implementations on the XMOS device and will try to investigate on how will these implementations behaviour when they go highly parallel. There would many hinder forces go against parallelization. This project is intend to detect these hinder forces and tries to see how can we avoid them or reduced the navigate effect of them. Also, this project will investigate on how to make compromise between the efficiency and the flexibility for a parallel implementation. The objectives of this project are showing below.

1. Implement several algorithms on the XMOS device.

2. Make the parallel implementations run on different number of cores and evaluate the results.

3. Demonstrate a situation in which an implementation could be slowed down while being over parallelized.

4. Detect the hindering forces against parallelization.

5. Investigate how to increase the implementation's scalability.

6. Discover guidelines about how to compromise between efficiency and flexibility.

7. Optimize and re-implement to obtain a better efficiency or flexibility.

1.2 Structure of this Dissertation

This dissertation is formed by four main parts.

The following chapter introduces the background of this study. This chapter briefly states the current situations for the field of parallel computation. Also this chapter reviews some parallel architecture, investigates some parallelizable algorithms and then introduces some methods to evaluate the parallel implementation.

Chapter 3 shows the development and procedure of this project. It describes the details of the parallel implementations, explains the choice between different core connection architectures and introduces the methods used to improve these implementations.

Chapter 4 demonstrates and analyses the results for all the implementations.

Finally, a critical evaluation of the research together with some suggestions on further work is given in chapter 5 and chapter 6.

Chapter 2: Background and Context

2.1 Why Parallel — Motivation

Both super computers and portable processors have increased their computation power a hundred times faster than just a decade ago. These increases will continue in the following years. However, new time consuming applications will always appear. They will easily overcome the increases and maintain the requirement for needing more computational power.

Historically, the improvements of computational power are mostly achieved by improving a single CPU's performance. It could be both improving the architecture of the processor and raising their clock speed. The clock speed of today's processors is trending to no longer any significant increases, which is because raising the clock speed lets the processors generate too much heat to dissipate.

$$\text{Power Consumed} = \text{Capacitance} \times \text{Voltage}^2 \times \text{Frequency}$$

Raising the clock speed will significantly increase the power consumption. This is because a processor that has a higher frequency also needs to be run at a higher voltage. So keeping the same computational power, having multiple core processors can reduce the power consumption. This solves the heat dissipation problem, saves power for super computers, and also stretches the battery life for portable devices.

The processor architecture's improving speed is also being limited. In the Electronic Design Automation industry, the next generation of processor is always designed by running the design tools on the newest machines [4]. This means the architecture of the next generation's single processor is always limited by the computational power of today's computers. Building multiple core processors can easily break this limitation.

2.2 Problems Faced and Interested Aspects for Parallel Computing

2.2.1 Parallelization in Real World

Parallelization does not only exist in the computer science area, it is already everywhere in our life. For example, the company has become the most ubiquitous organization operating in the world's economy today. The power of a corporation is far beyond what a single person can achieve. Companies combine single people together and tend to make them work in the most efficient way. People work in parallel inside a company. The manager partitions the company's goal into different parts and makes people work in their own domains. People communicate with each other, share their opinions and pass on the results. In this view, a well-managed company is quite similar to an efficient parallel computing system.

2.2.2 Efficiency Matters

For a computing system, what matters is how to do things efficiently. Efficiency determines whether things can be done in time. As a company or as a computational system, both can be eliminated when they fail to run effectively. Just like running a company well is a kind of art, there is a lot to explore to achieve an efficient parallel system.

2.2.3 Problems Faced in Current Parallel Computing System

Serial computing is a mature field and has made great contributions in changing our life for the better in the last few decades. As we turn to parallel computing in a moment, many difficulties have been placed in front of computer engineers.

There is no unified parallel architecture at the moment, which causes the following problems:

An algorithm optimized for one type of architecture cannot be easily applies to other architectures.
One parallel algorithm may be well adapted to a certain parallel architecture but rejects other architectures.
There are different programming languages for different parallel architectures.

These reasons have led to some unfortunate consequences:

For the programmer, it is difficult to determine which architecture is suitable for use. After choosing the architecture, the programmer may need to learn a new programming language.
For the development of parallel algorithms, by applying different architecture, the parallel algorithms will fall into different branches. They can hardly support each other.
For the development of parallel architecture, because of the lack of a unified parallel architecture, the development of parallel architecture will have some drawbacks.

This project intends to investigate certain parallel architecture and make an effort to defuse the consequences introduced above.

2.3 Architectures for Parallel Systems

Parallel computer architectures can be classified in many different categories based on different aspects, such as the interconnection networks used in the system or the memory types they apply. The following topics will provide a general background about parallel architectures.

2.3.1 Flynn’s Classical Taxonomy

There are different ways to classify parallel computers. Flynn’s Taxonomy is one of the most widely used classifications, and has been in use since 1966. [5] Flynn’s Taxonomy distinguishes parallel computer architectures into four processing modes, based on whether the processors execute single or multiple instruction streams at the same time, and whether the processor is processing single or multiple data. The matrix below shows the four classifications of Flynn’s Classical Taxonomy. [6]

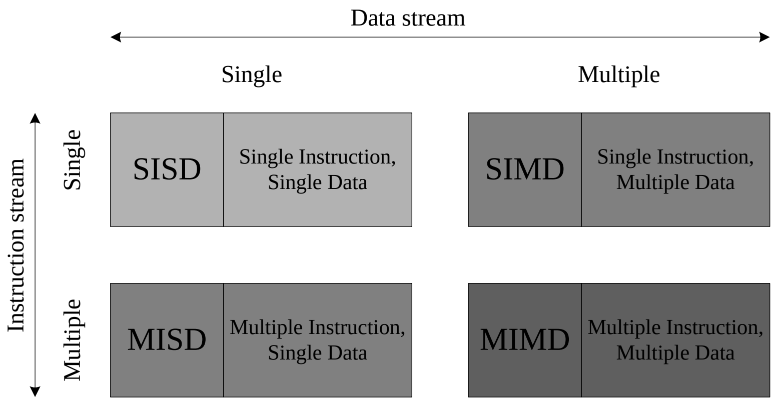

Data stream
Single
Multiple
Instruction stream
Single
Multiple
SISD
Single Instruction, Single Data
SIMD
Single Instruction, Multiple Data
MISD
Multiple Instruction, Single Data
MIMD
Multiple Instruction, Multiple Data

Figure 1, The 4 classifications of Flynn's Taxonomy. (reconstructed from page 55 of [12])

In most cases among Flynn's Taxonomy, MIMD can be used to represent parallel architecture. And SISD is the only category for serial computation. In this case, the comparison between serial and parallel computation is turned into the comparison between SISD and MIMD architecture.

1. Single Instruction, Single Data
SISD is a serial (non-parallel) architecture that can be represented by the classical von Neumann Architecture. It has a single CPU and a single memory unit connected by a system bus.

For a SISD machine, only one instruction stream can be processed by the CPU during any clock cycle, and only one data stream can be used as input during any clock cycle.

Until now, this was the most commonly used architecture. Most of today's micro-controllers and old computer with single core processors are using this architecture.

2. Single Instruction, Multiple Data

SIMD is a type of parallel architecture that has multiple processors. All processing units only can execute the same instructions synchronously (lockstep) at a given clock cycle. However, each processor can operate on their own data element. This makes the SIMD structure quite suitable for high regularity problems such as image processing or scientific applications, whose data sets can be partitioned easily.

3. Multiple Instruction, Single Data

Corresponding to the SIMD, MISD is another example of multiple processor architecture whose processing units can have independent instruction streams executed on a single data stream. Only a single data stream can be fed into those processing units.

4. Multiple Instructions, Multiple Data

The MIMD architecture can have multiple processors executing different instructions on different data streams. There is no common clock for the entire system, and the execution can be synchronous or asynchronous.

Among Flynn's Taxonomy, MIMD have the most flexibility in partitioning data and instructions to different processors. It has the best adaptability to various algorithms, which makes MIMD the most likely to become the most common type of parallel computer.

2.3.2 Memory Type for Parallel Computers

The memory architecture for parallel computers mainly falls into two categories: Shared Memory and Distributed Memory. There's also a type called Hybrid Memory, which combines the architectural characteristics of both Shared Memory and Distributed Memory.

1. Shared Memory

The most significant characteristic for a shared memory parallel computer is all of its processors have the ability to access the entire memory as a global address space. When a processor writes to a memory location, the change is visible to all other processors. Shared memory machines have two main classes: Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA). The structure for UMA and NUMA are shown in the figure below.

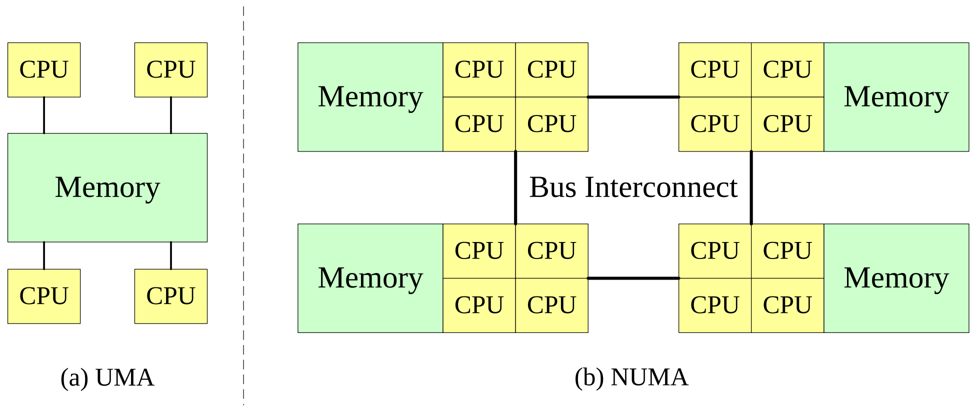


Figure 2, The UMA and NUMA classes (reconstructed from Chapter 4 of [6]).

UMA and NUMA are divided based on whether the processors have equal access time to all memories. UMA has equal access times while NUMA does not.

2. The Advantages and Disadvantages for Shared Memory Architecture

The memories on shared memory architecture are more close to CPUs than the distributed memory architecture so the processors are quicker in accessing or sharing data. The primary disadvantage for shared memory architecture is the lack of scalability. Adding more processors will geometrically increase the interconnect path between CPUs and memory. Also, the memory space changed by one processor will affect all the other processors. The programmer must make sure the memory spaces have been accessed by the processors in the correct order.

3. Distributed Memory

Parallel computers with distributed memory have distributed processors that work independently. Each processor has its own local memory. One processor's memory address cannot be mapped by another processor directly. Also, one processor changing its local memory will have no effect on the other processor. The most common characteristic shared by distributed memory systems is they have a communication network to connect the processors together. When a processor needs the data from another processor, a communication link is usually built and the data can then pass from one processor to another.

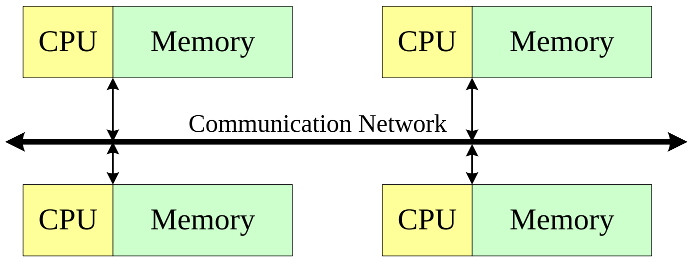


Figure 3, Distributed memory class (Reconstructed from Chapter 4 of [6]).

4. The Advantages and Disadvantages for Distributed Memory Architecture

The primary advantage for distributed memory architecture is the memory and processors are scalable. The processor number and the memory size can increase proportionately. Unlike the shared memory, the processor in distributed memory architecture only can access its own memory. The programmer does not need to worry about interference between processors while accessing memory.

The disadvantage is the programmer needs to handle the data communication between processors. Also, as distributed memory architecture does not have global memory, it is difficult to use existing data structures on such architecture.

2.3.3 XMOS Architecture Review

In this project, we chose to investigate parallelization on XMOS devices. XMOS is a conventional multi-core hardware that is based on the message passing architecture with distributed memory. A 64-core XMOS device has been selected, which can have up to 512 threads working at the same time. For such a portable device, having 512 concurrent threads is a relatively large number and it is sufficient for us to scale our algorithms on it.

1. XS1 Family

The XMOS devices used in this project are the 4-core XS1-G4 device and the 64-core XMP-64 device. Both of these devices belong to the XMOS XS1 product family. The XS1 family contains the high-performance G-series, which are fabricated on TSMC's 90nm process technology, and the energy-efficient L-series, which uses 65nm process technology. [8]

Device	XS1-L1	XS1-L2	XS1-G2	XS1-G4

Xcores	1	2	2	4
Threads	8	16	16	32
MIPS(max)	400/500	800	800	1600
SRAM(total)	64 Kbytes	128 Kbytes	128 Kbytes	256 Kbytes
OTP(total)	8 Kbytes	16 Kbytes	16 Kbytes	32 Kbytes
I/O	3v3	3v3	3v3(5v tolerant)	3v3(5v tolerant)
Power consump-tion	15-200mW	30-400mW	200-700nW	200-1200mw
Packages(I/O)	QFP48(28) QFP64(36) QFP128(64)	QFN124(84)	BGA144(88)	BGA144(88) BGA512(256)

Table 1, The XS1 family specification.

The high-performance XS1-G series devices are intended to contain larger systems in a signal chip, while the energy-efficient XS1-L series devices are optimized to deliver a significant amount of computing performance with low power consumption.

2. Scale up the XS1 Architecture

The XS1 family of devices are multi-threaded processor architectures formed by different numbers of XCore™ processors connected by the communications link. The architecture is scalable and any number of XCore™ processors can be connected together. [8] The 4-core device XS1-G4 combines four XCore™ processors together. The four processors are connected to communication via a high-performance switch. They also can communicate with other XCore™ processors by inter-chip links. Each XCore™ has its own memory and is placed on a single chip. Each core can support a maximum of 8 threads.

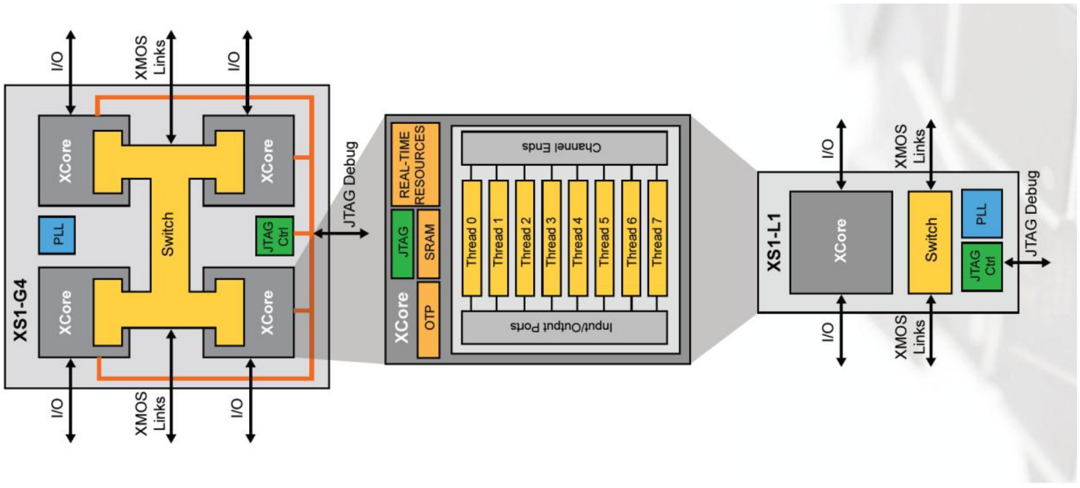


Figure 4, The XS1 family architecture (from page 1 of [8]).

The 64-core device XMP-64 is a scaled up version of the XS1-G4. It connects 64 XCore™ processors together on a single PCB, each with four processors packaged in a XS1-G4 device. The XMP-64 board can support 64*8, which is 512 threads. It is a massive parallel device used to demonstrate the scalability of the XS1 architecture.

3. Architecture Category for XS1 Family

The XS1 family is distributed memory MIMD architecture using the message passing model. Each XCore™ processor has its local memory. There is no global memory address for all processors. One processor cannot directly map to another processor's memory. When data needs to be passed between processors, a channel is generated between these two processors. The data could then be passed through the channel. The data transfer needs a send operation on the sending processor and a matching receives operation on the receiving processor.

4. Communicate Data via Registers and Shared Memory in the XS1 Family

Usually we assume passing data between processors is always through the channels. However, threads on the same XCore™ processor can also communicate data via registers and shared memory. But the programmer needs to make sure that the threads are synchronized in order to avoid race conditions. The number of channels that can be allocated are limited for each processor. Communication via channels also takes more time than communication via registers. As a result, communication via registers and shared memory is quicker and can increase the programming flexibility, but it will give more challenges to the programmer and will reduce the scalability of the program.

2.4. Algorithms with Potential to Run in Parallel

Algorithms play an important role in the field of computer science. There are various parallel algorithms already in existence in different types of architecture. And there are many more serial algorithms waiting to be translated into parallel.

2.4.1 The Legacy Algorithms from Serial Computation

There are a large number of brilliant algorithms that have been optimized for serial computation in the last half century. As parallel computation becomes more and more popular today, those serial algorithms have become a great legacy for parallel computation. However, a great algorithm for serial computation may or may not be suitable for parallel computation. In other words, a certain kind of algorithm may fit one kind of computer architecture well, but be rejected by another kind of computer. By implementing various algorithms on the XMOS devices, we can derive some guidelines for how algorithms fit an architecture and how to optimize them.

2.4.2 The N-Queens Problem

Some algorithms are naturally more suitable for parallelization than other algorithms. The N-Queens Problem is one of them which have a good potential to be translated into parallel. The N-Queens Problem is the problem of putting N chess queens on an N×N chessboard and ensuring no two queens attack each other. The number of solutions for the N-Queens Problems on each board size is stationary values.

Board Size (length of one side of N x N chessboard)	Number of Solutions to N-Queens Problem
1	1
2	0
3	0
4	2
5	10
6	4
7	40
8	92
9	352
10	724
11	2680
12	14200
13	73712
14	365596
15	2279184
16	14772512
17	95815104
18	666090624
19	4968057848
20	39029188884
21	314666222712
22	2691008701644
23	24233937684440

Table 2, Number of solutions for the N-Queens Problem. [15] [13]

The number of solution to N-Queens Problem which is list in the table above can be used to check the correctness of the implementation. As shown in the table, the number of solutions for the N-Queens Problem can have an exponential growth with the increase of the board size. This project only evaluates the N-Queens Problem implementation on the XMOS device with the board size from 9×9 to 18×18. That is because the problem size of the N-Queens Problem with a board size 18×18 is sufficient for the XMOS device.

There are many ways to store the board image of the N-Queens Problem. In this project we chose to use bitmaps to represent the board image. [13]

- - - - - Q - -	00000100	0: Start
- - - Q - - - -	00010000	1: |
- - - - - - Q -	00000010	2: |
Q - - - - - - -	10000000	3: | Back Tracking
- - - - - - - Q	00000001	4: |
- Q - - - - - -	01000000	5: |
- - - - Q - - -	00001000	6: |
- - Q - - - - -	00100000	7: Last

Table 3, The board image represented by binary numbers.

Table 3 uses eight 8-bit binary numbers to store the 8×8 chessboard. 0 represents the empty space on the chessboard and 1 represents the queens. In this case, we only need N integers to store an $N\times N$ chessboard. The board size could be up to 32×32 as the integers on the XMOS device are 32 bits. Using the bitmap to store the chessboard can save lots of memory, rather than using two-dimensional arrays. This is particularly useful while doing implementation on the XMOS architectures, which have very limited memory.

2.4.3 Sorting Algorithms in Serial

The sorting algorithm is very famous among computer scientists. It is usually considered as the fundamental problems in the study of algorithms. It is used when we need to sort a set of numbers into non-decreasing order. The sorting problem can be formally defined as:

Merge sort and insertion sort are two simple algorithms easy to start. Even though these two algorithms are quite basic, each of them shows some extreme features when translating them into parallel. Those features are worth studying and will give us a general view of how algorithms' characteristics are affected when translating an algorithm into parallel.

Input: A sequence of n numbers $<a_1, a_2, \ldots, a_n>$.
Output: A permutation (reordering) $<a'_1, a'_2, \ldots ,a'_n>$ of the input sequence such that $a'_1<=a'_2<= \ldots <=a'_n$. [9]

When implemented on a serial computer, the running time for insertion sort is $O(n^2)$ and the running time for merge sort is O(n lg n) [9]. The insertion sort can be quicker when the input size is very small. The merge sort will beat insertion sort when the input size is big enough. As serial algorithms, we can say the merge sort is generally smarter than the insertion sort. However, when we turn to parallel architecture, there would be a huge difference between these sorting algorithms.

1. Insertion Sort in Serial

Insertion sort is an efficient algorithm for sorting a short array of elements. It works like sorting a hand of playing cards. We take one card at a time from the table and insert it into the correct position among the holding cards. When we get a new card, we compare it one by one to the holding cards to find the correct position. For the convenience of implementing the sorting algorithm, we always start the comparison from the right side of our holding cards, as shown in the figure below. [9]

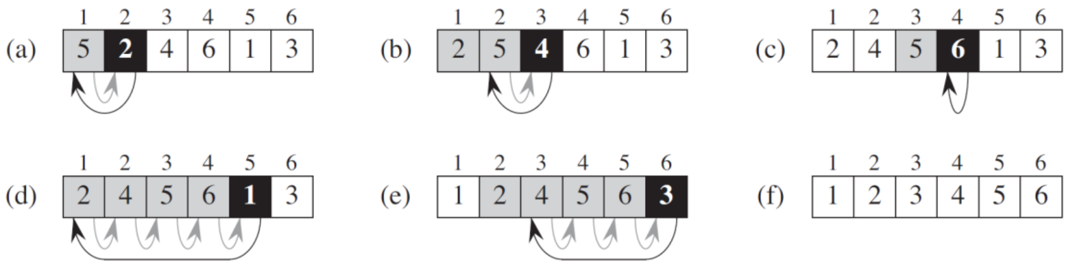


Figure 5, The operation for insertion sort (from page 18 [9]).

The figure above shows the procedure sorting the array A = <5, 2, 4, 6, 1, 3> using insertion sort. The value in the black rectangle is the new card we take from the table, and we call it the key. All the values to the left of the black rectangle correspond to the holding cards, which are always sorted before we take a new card from the table.

There is a pseudo-code called INSERTION-SORT present in the procedure for the insertion sort.

```
INSERTION-SORT(A)                                [9]
for j = 2 to A.length
    key = A[ j ]
    // Insert A[ j ] into the sorted sequence A[ 1 .. j - 1 ].
    i = j - 1
    while i > 0 and A[ i ] > key
        A[ i + 1 ] = A[ i ]
        i = i - 1
    A[ i + 1 ] = key
```

2. Merge Sort in Serial

Merge sort is a little more complex than insertion sort. We can start to understand it from its pseudo-code. The merge sort is achieved by using a recursion function named MERGE-SORT. The function MERGE-SORT contains a function MERGE that is used to merge two sorted sequences together. [9]

```
MERGE( A, p, q, r )
$n_1$ = q - p + 1
$n_2$ = r - q
```

```
let L[ 1 .. $n_1$ + 1 ] and R[ 1 .. $n_2$ + 1 ] be new arrays
for i = 1 to $n_1$
    L[ i ]=A[ p + i - 1 ]
for i = 1 to $n_2$
    R[ j ] = A[ q + j ]
L[ $n_1$ + 1 ] = $\infty$
R[ $n_2$ + 1 ] = $\infty$
i = 1
j = 1
for k = p to r
    if L[ i ] $\leq$ R[ j ]
        A[ k ]=L[ i ]
        i = i + 1
    else A[ k ] = R[ j ]
    j = j + 1
```

The function MERGE is used to merge two subarrays into a single sorted subarray. In the MERGE(*A, p, q, r*), *A* is an array, *p, q, r* points to a position in array *A* and $p \leq q < r$. While calling the MERGE(*A, p, q, r*) function, subarrays *A[p .. q]* and *A[q + 1 .. r]* should always in sorted order. It merges the two subarrays and the result replaces the current subarray *A [p .. r]* into a sorted order. More details about this function can be found in Chapter 2 of the reference book [9].

```
MERGE-SORT( A, p, r )
if p < r
    q = [ ( p + r )/2 ]
    MERGE-SORT( A, p, q )
    MERGE-SORT( A, q + 1, r)
    MERGE(A, p, q, r)
```

After understanding the sub function MERGE, we can see the recursion function MERGE-SORT clearly shows the procedure of the sorting algorithm. To solve the input sequence A = <A[1], A[2], … , A[n]>, we set the initial call as MERGE-SORT(*A, 1, A.length*). The recursion function then starts to divide the input into subarrays until there is only one value in the subarray that is obviously sorted. Then the function starts to merge the sorted subarrays together until we get the entire sorted array. The operation of the merge sort is shown in the following figure.

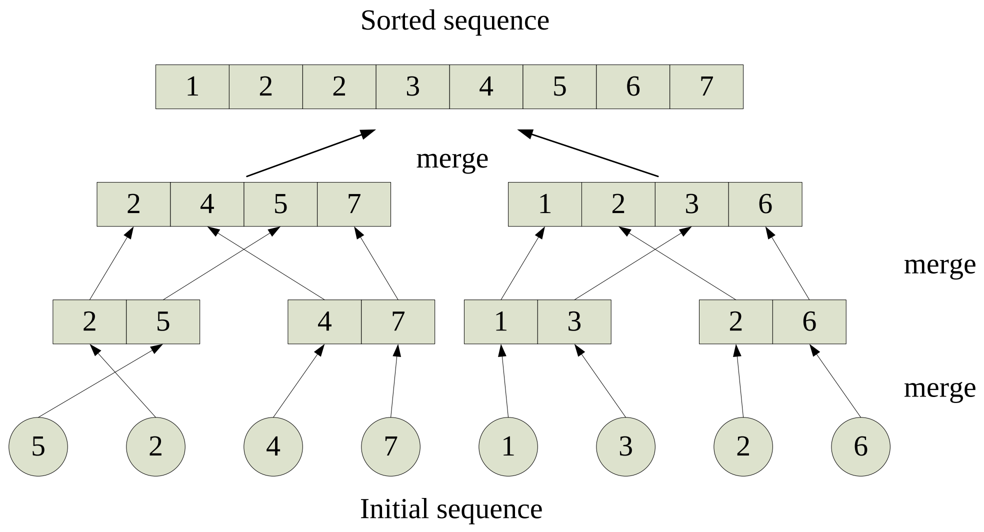


Figure 6, Operation of merge sort (from page 35 [9]).

2.5 How to Evaluate the Performance for Parallel Computation

This section introduces a set of selected metrics, which are criteria used to evaluate the performance for parallel computation. The metrics for parallel computing are highly problem oriented. One problem may give a significant good performance on certain architecture types, but poor performance on other architectures.

1. The Frequency of Data Communication [2]

In parallel computing, the frequency of data communication can be described as frequent (high) or infrequent (low). Frequent communication is like a section of code requiring data to be passed between processors in every line. Conversely, an algorithm that has infrequent communication may have most of its subroutines executed locally on each processor without data communication.

For the N-Queens Problem described in the former chapter, when it was partitioned and run on multiple processors, each partition are able to run independently until returning the result. In this case, the N-Queens Problem can be described as having an infrequent data communication. In the other hand, the Merge Sort will have to handle with a large number of data. When the Merge Sort is run in parallel the entire input will need to be passing through processers. This makes the Merge Sort having a frequent communication.

2. Locality

Access to a remote memory space is more expensive than access to local memory. It takes a longer time and occupies more resources, such as the I/O mechanism. Local memory access does not need to use the network and it can be done in 10s of cycles, whereas the remote memory access involves the network and takes 100s or 1000s of cycles. Hence locality has become an important property for parallel algorithms. It reflects the algorithm's ability to be run efficiently in parallel.

3. Speedup

Speedup is a simple and widely used indicator to measure the performance of an optimized code. A program's speedup can be defined as

$$\frac{T_{before}}{T_{after}}$$

Where T_{before} is the running time before optimization and T_{after} is the running time after optimization. As in this project most of the comparisons are between the serial implementation and parallel implementation, T_{before} can be considered as the running time before parallelization, which is the running time for serial implementation. And T_{after} can be considered as the running time for parallel implementation.

4. Scalability

The scalability for a parallel system indicates its ability to generate a proportionate speedup while scaling the implementation. The implementation can be scaled by either running on more processors concurrently or having a larger size of input data.

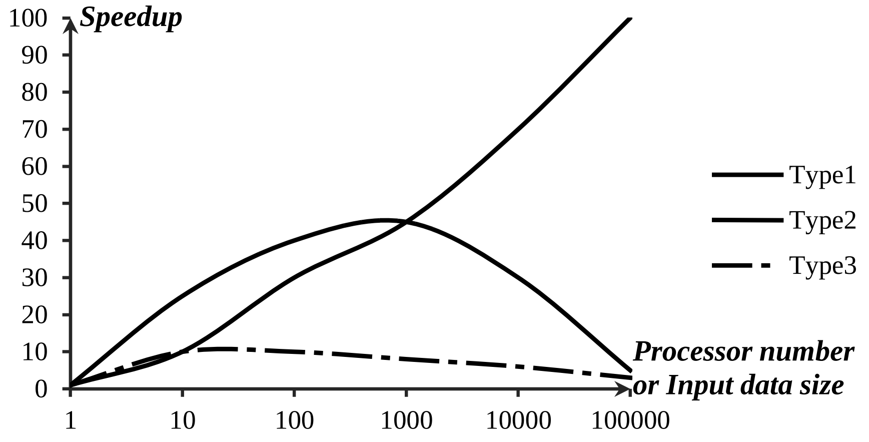


Figure 7, Imaged scalability characteristics for three types of algorithms.

The figure above shows the scalability characteristics for three imagined algorithm types. In this case, we can say type1 algorithms have very good scalability. Type2 algorithms have a good scalability when the processor number or input data size is less than 1000. Type3 algorithms have very bad scalability. Type2 algorithms are parallel overhead when the processor number or input data size is greater than 1000.

5. Amdahl's Law

In 1967, Gene Amdahl introduced an efficient way to measure a parallel problem's scalability, which is known as Amdahl's Law [10] [11]. Amdahl's Law defines a program's potential speedup by the size of the program's fraction (P) that can be run in parallel.

$$speedup \quad = \frac{1}{P/N + S}$$

Where P is the fraction of the code that can be run in parallel, N is the number of processors and S is the serial fraction of the code.

As a program can only be divided into two fractions, the P fraction and S fraction, the serial fraction is equal to (1 – P). The following graph is generated according to Amdahl's Law by setting the parallel fraction value P as 0.50, 0.90 and 0.99. We can see how the speedup increased as the processor number exploded by having the three parallel fraction values.

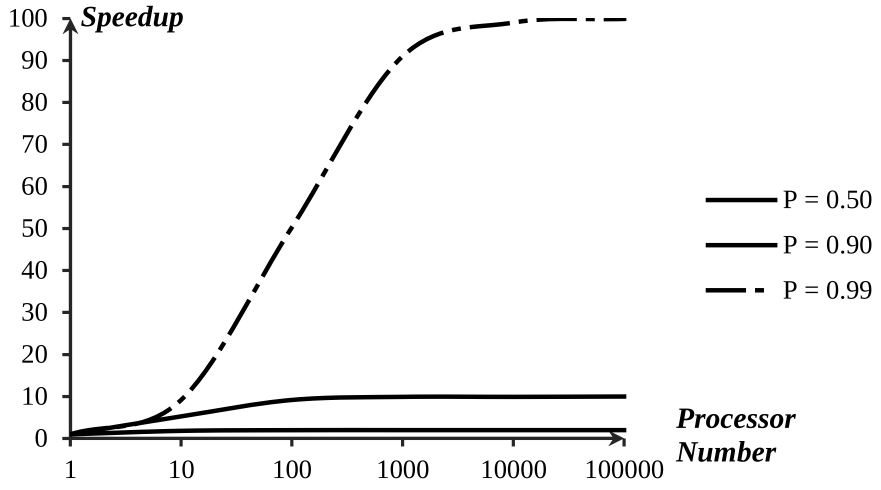


Figure 8, The expected scale - speedup generated according to Amdahl's Law.

The above graph shows that the P fraction of a code is a very important property that indicates the scalability of a parallel program. A program with a larger P fraction can get a great speedup while processor numbers increase.

For some algorithms, such as the divide and conquer kind of algorithm, the parallel fraction of the code always increases quicker than the serial fraction. That is because those kinds of algorithms only have certain processes that can be done in serial. For such algorithms, increasing the problem size will increase the parallel fraction of the program, and it will finally lead to increasing the scalability of the algorithm in increasing processor numbers. We can say problems that increase the percentage of parallel fractions with their size are more scalable.

2.6 Conclusion

As it is described in this review, the trend for computation is turning from serial to parallel. Parallel computing techniques have been widely adapted to every aspect of computation for optimizing speed and efficiency. However, there are more implementations waiting to be done in parallel. As we can see, various parallel architectures exist at the moment. It is important to know some principles to choose a parallel architecture when we want to implement an application in parallel. Countless algorithms will be designed in parallel, or will be translated from serial into parallel in the following years. There is a great need for guidelines that can help when translating algorithms from serial into parallel. To get a view of these principles and guidelines, this project will investigate parallel implementations. Two XMOS devices are used in this project to investigate the scalability and parallelization efficiency for our implementations.

Chapter 3: Project Development

3.1 Purpose of the project

This project will do some parallel implementation on the XMOS devices and try to see how those implementations behave when they go highly parallel. The purpose of doing this is to predict the difficulties that will be faced on our path to parallelization.

The implementations in this project will show how the computational performance changes with the growth of core numbers. The increase of computational performance will not always be ideal. That is because there are many hindering forces against parallelization that will slow down the computational progress. The implementations in this project should able to elicit those hindering forces.

When going more parallel, those hindering forces can make the improvement of computational powers less impressive. In some cases, it can even slow down the progress of computation. This means in some situations using more cores will not significantly speed up the progress, which makes going more parallel pointless. This project should be able to demonstrate these situations so we can determine when we should stop going more parallel.

Efficiency is not everything in the field of parallel computation. The flexibility of the parallel code is also important. A flexible code can be easy to transfer to run on a different number of cores. Some code can make the most use of any number of cores, but some can only run on a fixed number of cores. This project will show how important flexibility is by showing two very different types of code.

3.2 Description of the Work

There are a number of implementations in this project used to do the evaluations. These implementations can be distinguished into two types based on the different algorithms being implemented, which are Merge Sort and The N-Queens Problem. These two algorithms are only used as the bottom element of the parallel implementations, and we want them simple and representative. For parallel implementations, managing the data flow and the priority of parallelization is much more complex than the implemented algorithm itself.

The first thing to do is translate the two algorithms in serial from C to the XC language. The XC language is slightly different from C; for example, it cannot use pointers but can pass data through channels between cores [14].

Then we implement Merge Sort in parallel and make it run on a different number of cores. The Merge Sort in parallel is designed to fit the nature of the Merge Sort algorithm as much as possible to generate more efficiency, but it will lack of flexibility. After doing the evaluations, the parallel Merge Sort has been optimized to make some improvements by knowing its shortcomings. After all, the Parallel Merge Sort has been deeply optimized on a 4-core device, which is the XC-1A kit. We are implementing the Merge Sort on both the XMOS XC-1A Development Kit and the XK-XMP-64 Development Board in both serial and parallel. Even though all the jobs can be done on the XK-XMP-64 Board, we are using the XC-1A kit as much as possible because the tasks of building the binary files for the XK-XMP-64 Board or burning the binary files onto it are very time consuming.

The parallel implementation for the N-Queens Problem has been done in a very different way. We first implement the algorithm on the XC-1A Development Kit and we will pretend the core number is unknown. When the implementation runs properly on the XC-1A kit, it can be transplanted to the XK-XMP-64 Board by simply changing the defined core numbers. The structure of the N-Queens Problem in parallel has more flexibility than the structure of the Merge Sort in parallel.

All the implementations will be run on a different number of cores with different input sizes.

3.3 Parallel implementations for Merge Sort

3.3.1 Serial implementation of Merge Sort on XMOS (limitations of XMOS)

As described in Chapter 2.4.3, the serial Merge Sort algorithm is achieved by using the recursive function. The figure below shows the structure of serial implementation for Merge Sort. In Merge Sort, each node of the structure means a call of the MERGE-SORT function.

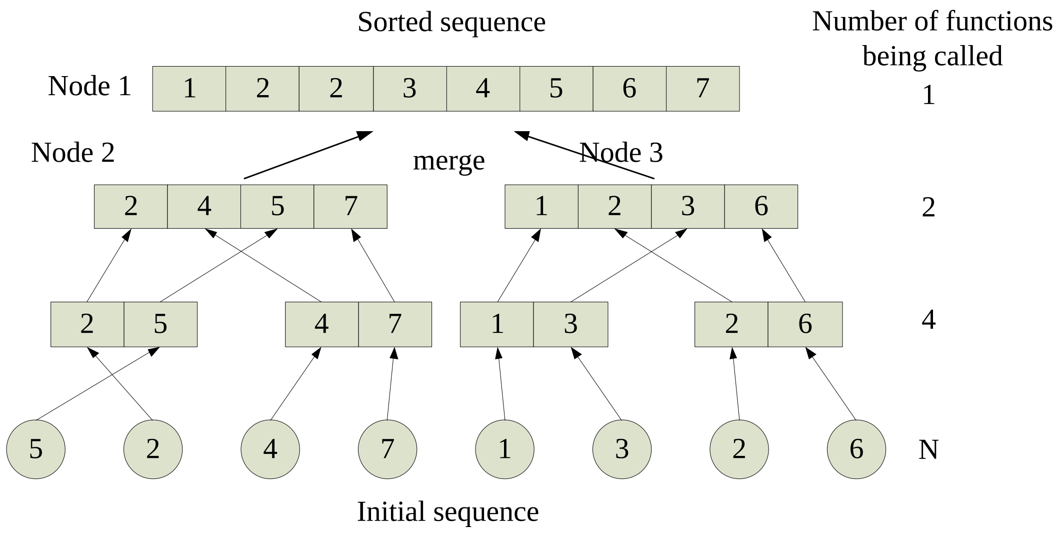


Figure 9, Number of functions being called by Merge Sort (from page 35 [9]).

Thus, when we sort N integers using Merge Sort, the recursive MERGE-SORT function will be call 2N-1 times. For example, if we want to sort 800 numbers the MERGE-SORT function will be call 1599 times. Let us assume there are five 32-bits of data (20 bytes) needed to be stored on the stack for each called function; 31980 bytes would be needed for sorting 800 integers. However, there is only 64 KB SRAM memory available for each XS1-G4 core (see Chapter 2.3.3 on page 8). We still need memory to store the input array and the subarrays used during the sorting process.

In practice, the maximum number of integers that can be sorted by Merge Sort on a single XMOS core is about 700. This number is far from enough as we will finally let the algorithm run on the 64-core device, which can have a maximum 512 threads running simultaneously.

Two methods are used here to overcome the limitations of the hardware. Firstly, when the subarrays become short enough during the recursive separating operations, we stop calling the MERGE-SORT function but use the Insertion Sort algorithm to sort the subarray. This is a widely used method as Insertion Sort can be quicker than Merge Sort when the input size is small. Insertion Sort is not a recursion function so it can reduce the number of recursion calls. In this case, we use Insertion Sort when the subarray size is equal or less than 4. This number is chosen because it can significantly reduce the stack use of Merge Sort while still keeping the characteristics for Merge Sort.

The original recursion calls for sorting N integers using Merge Sort are:

$$2N - 1 \cong 2N$$

When implementing Insertion Sort when the subarray size is equal or less than 4, the number of recursion calls has been reduced to:

$$2 \times \frac{N}{4} - 1 = 0.5N - 1 \cong 0.5N$$

The stack usage caused by calling the recursion function then can be reduced three-quarters.

The second method is to use a short integer instead the integer to store inputs. This will limit the inputs but will have little effect on our evaluations. This will halve the size of the memory used for storing the input array and the subarrays.

Finally, the optimized Merge Sort running on a single XMOS core can sort more than 6000 short integers.

3.3.2 The development of Merge Sort Implementation on XK-XMP-64

After the serial code has been polished, it becomes a kind of standard for the parallel implementation of Merge Sort. When writing the parallel Merge Sort code, we do our best to keep the characteristics of the Merge Sort algorithm to avoid making the parallel Merge Sort a completely different code than the original Merge Sort algorithm.

The most ideal situation for parallel Merge Sort would be having a core with a large number of threads, and when the MERGE-SORT function calls itself, the new function will run on a new thread. However, we can only have a maximum of 8 threads run simultaneously on an XMOS core.

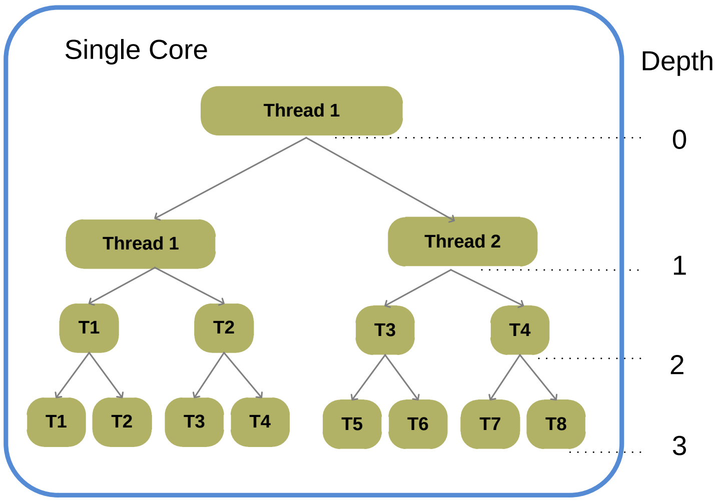


Figure 10, Merge Sort implementation on a single core.

As shown in figure 10, the parallel Merge Sort implementation only goes three levels deeper on a single XMOS core. When Merge Sort goes to a new level, the threads for the old levels will be stored on the stack and are shut down, so they would not be count as simultaneously running threads.

One solution for when the threads run out is to connect two threads from two new cores to the lowest level threads, as shown below.

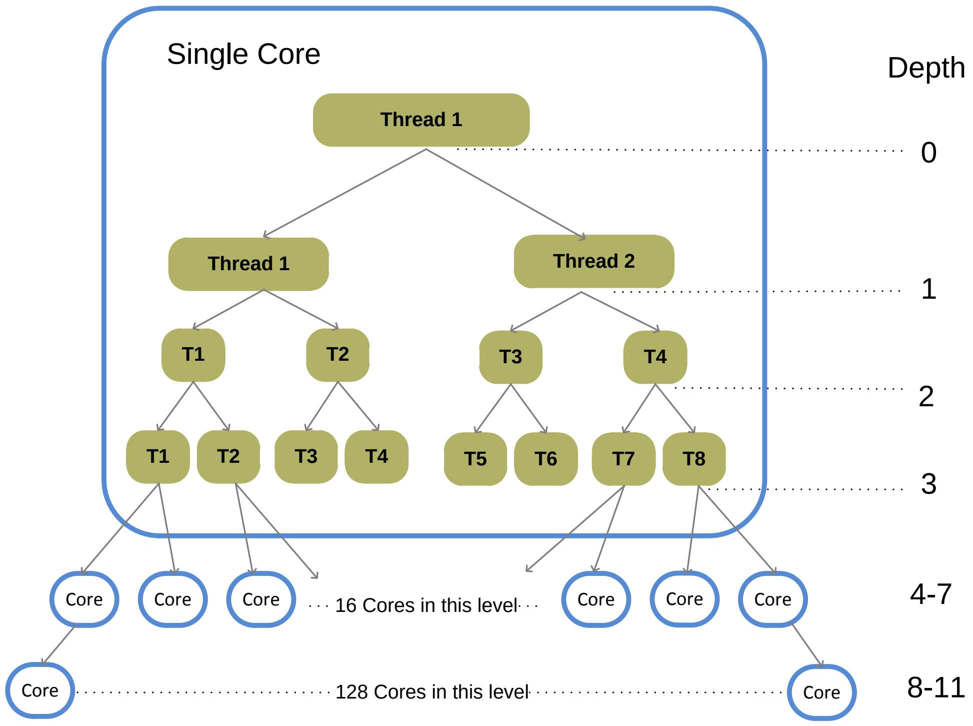


Figure 11, Core connection architecture for Method 1.

Method 1

As shown in the figure above, each core can let the Merge Sort go three steps deeper. And finally they will have 8 threads running simultaneously on each core. Each of the 8 threads will be connected to a new core as the current one has run out of threads.

In this case 16 more cores will be needed if we want go one step deeper from depth 3 to depth 4, and 128 cores will be needed to go from depth 7 to depth 8. This solution is extremely lacking in flexibility. It can only make use of 17 cores for the 64-core XK-XMP-64 Development Board and can only make Merge Sort go to depth 7 on the 64-core device.

In order to make the most use of the 64 cores and make Merge Sort have a deeper depth, the following core connection has been designed.

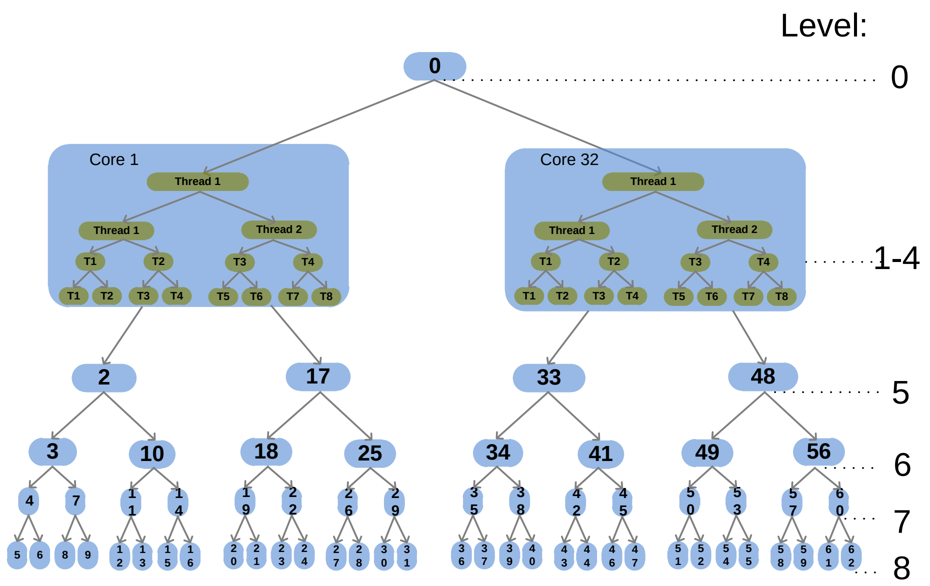


Figure 12, Core connection for Method 2.

Method 2

Figure 12 shows a new core connection for the parallel Merge Sort implementation. In this new structure, Merge Sort starts from a single thread on Core 0. The initial thread will not create new threads on Core 0, but will directly let the two new recursion calls run on Core 1 and Core 32. There are many reasons for why we do not make the full use of Core 0. The most important one is Core 0 needs to have static arrays to store the whole input data. Even when we only use a single thread on Core 0, it will have to store the whole input and two subarrays in order to store the two parts of the input separated by the Merge Sort algorithm. If we use more threads on Core 0, the memory of Core 0 will become a limitation for the maximum input size.

Also, if we have more threads on Core 0, we will need more cores on the next level. The structure would lack flexibility, just as the former one in figure 11.

There is another improvement in this new architecture: there are only two new cores connected to a higher level core. That is achieved by using the connect method showing below.

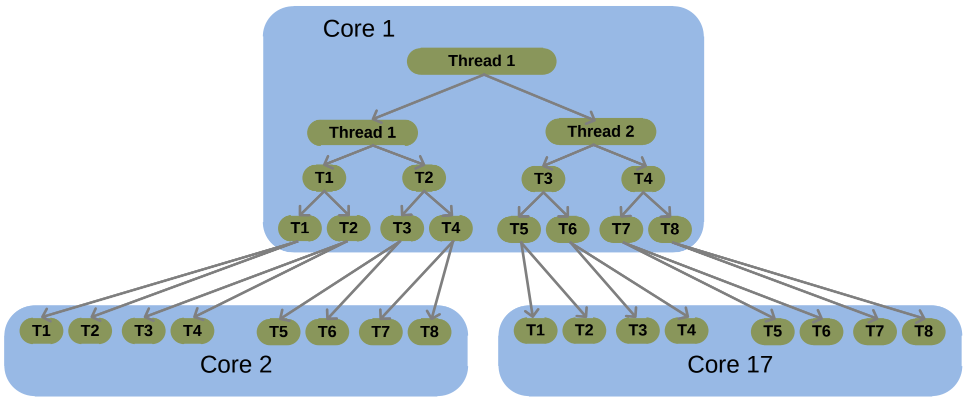


Figure 13, Detail of the connection for Method 2.

In this new implementation, when a core reaches its 8 thread limitation, each thread will produce two new threads on a lower level core. Thus there would be 16 new threads created by one higher level core. In this case, only two lower level cores need to be attached to one higher level core to make the Merge Sort one step deeper.

Each new thread in Core 2 and Core 17 can be activated by receiving a piece of data. These threads are starting with the following piece of code.

```
        cin :> data.len;
        cin :> data.lvl;
        for(i=0;i<=data.len;i++)
        {
                cin :> data.data[i];
        }
```

Threads in lower level cores will wait until the data package from higher level cores passing to it and then start execution. At the end of the execution, the modified data package will be passing to the next level of cores by the following code.

```
        //passing data to the next level of cores...
        cout <: len;
        cout <: lvl;
        for(i=0;i<=len;i++)
        {
                cout <: data[i];
        }

        // pending the results...
        for(i=0;i<=len;i++)
        {
                cout :> data[i];
        }
}
```

After sending data to the next level of cores, the current thread will waiting until the results passing back from the lower level cores.

	Method 1	Method 2
Depth of Merge Sort	Core Request	Core Request
1	1	3
2	1	3
3	1	3
4	17	3
5	17	7
6	17	15
7	17	31
8	145	63
9	145	127
10	145	255
11	145	511
12	1169	1023

Table 4, The number of cores required for difference depth of Merge Sort.

Table 4 shows how many cores are requested for both methods to let the Merge Sort go to a particular depth. Method 2 has better flexibility than Method 1. For Method 1, the requested core numbers will increase by 8 to the power of n for every 4 levels deeper. For Method 2, the requested core numbers will increase by 2 to the power of n for every level deeper. For some depths of Merge Sort, such as depth 7, 10 and 11, Method 1 requests a fewer number of cores than Method 2. But in other cases, Method 2 needs a fewer number of cores to achieve the same depth.

Fortunately, Method 2 can fit the 64-core device well by letting the Merge Sort go to depth 8 by using 63 cores. This is why we chose Method 2 for our parallel Merge Sort implementation. However, if the device has a core number such as 60 or 500, we cannot make the full use of the device with either of the two methods.

Some compromise has been made for the Method 2 shown in figure 12. Too much memory has been used in Core 1 and Core 32, and this will limit the maximum input size of the parallel Merge Sort implementation. In the original Method 2, Core 1 and Core 32 starts working when Core 0 passes all the data to Thread 1 from each of them. Thread 1 will then make the recursion call to create the new Thread 1 and Thread 2. The new threads will keep doing the recursion call until the 8 threads on each core have been used up. By doing so, the Merge Sort has separated its input data by four times; it can also be said the Merge Sort has gone to depth 4.

Each time the Merge Sort goes one step deeper, a set of subarrays will be required to store the whole input data which has been separated into different sizes. In the original Method 2, the Merge Sort goes from depth 1 to depth 4 on the second level nodes, which are Core 1 and Core 32. In this case, the input data will require being stored at least four times on Core 1 and Core 32. It will make the memory size of Core 1 and Core 32 a limitation for the whole system. The maximum number of inputs will be reduced.

Even worse, the XC language does not support dynamic memory allocation. So if we want to write a simple recursion function for it, all the subarrays will have to be set to the size of the longest one. That will cause all the subarrays to take 15 times the memory of the initial input. Otherwise, we can assign a specific size to the subarrays in each level of the Merge Sort depth. However, it will break the recursion structure and will cause the code to lose simplicity.

To overcome this limitation, Method 2 has been changed as shown below.

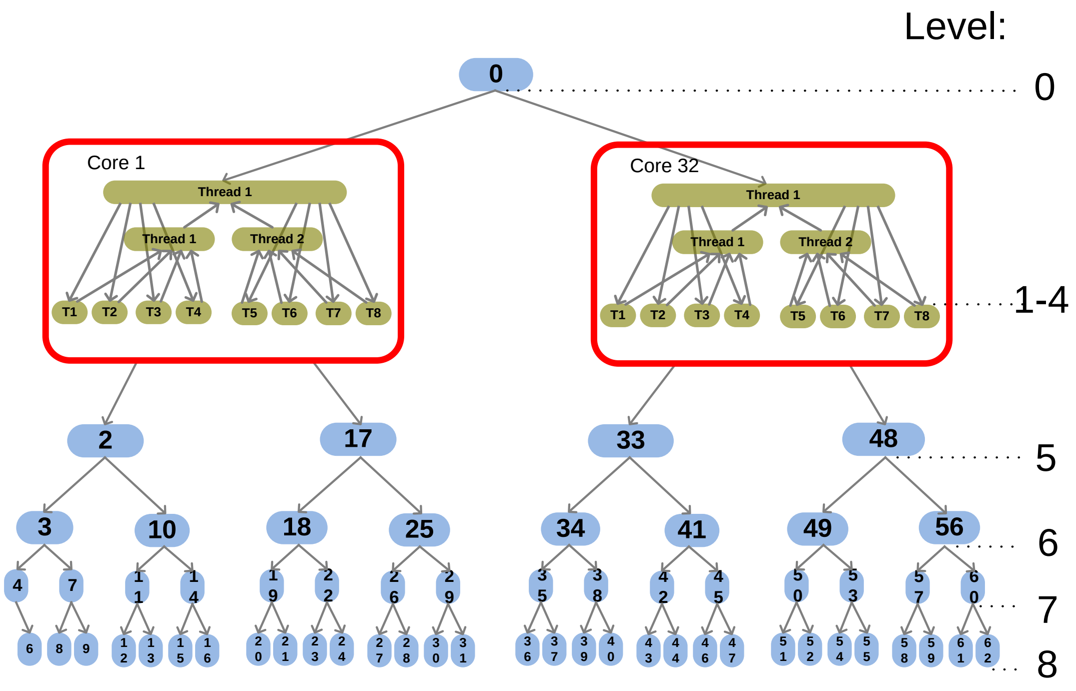


Figure 14, The improved architecture for Method 2.

The changes of Method 2 have been shown in the red frames of figure 14. Other than the original Method 2 shown in figure 12, Thread 1 in Core 1 and Core 32 will separate the data into 8 parts directly and pass them to the new threads. When all parts of the data have been sorted and returned to Core 1 and Core 32, each four of the subarrays will be merged together into two larger arrays, and the two larger arrays will be merged into the initial arrays in Thread 1. The MERGE function shown on page 12 is able to merge the two arrays together. Here, a MERGE4 function has been designed to merge four arrays together directly. It is based on similar logic as MERGE, and the code can be found in the appendix 8.8. By doing this, Core 1 or Core 32 only needs to store 1.5 times the size of the initial input on each of them. As Core 0 has to have arrays to store 2 times the initial input, the new method will release Core 1 and Core 32 from restricting the maximum input size.

3.3.3 Optimize the parallel Merge Sort on a 4-core device

The parallel Merge Sort implementation takes into account efficiency and flexibility. It is also worth seeing how the implementation be optimized on a particular device while giving up the flexibility. The XMOS XC-1A Development Kit is a 4-core device and can have a maximum of 8 threads run simultaneously on a single core. We are going to develop a deeply optimized Merge Sort algorithm on the XC-1A kit, and will see the gains and losses for such an implementation.

The first point for the optimization is the data flow. It seems the implementation could be more efficient if each of the cores has the same amount of work to do, otherwise one core could be waiting for another and the computational power shall be lost. For the Merge Sort on a 4-core device, we want to assign each core the same amount of numbers for it to sort with. Thus there would be a large amount of data needing to be transferred to each core, and each core will receive the same amount of data.

One way to spread the input data to each core is to pass them directly from Core 0. However, the XC language does not allow different threads to assess the same array at the same time; the data needs to pass to each core one by one as shown below.

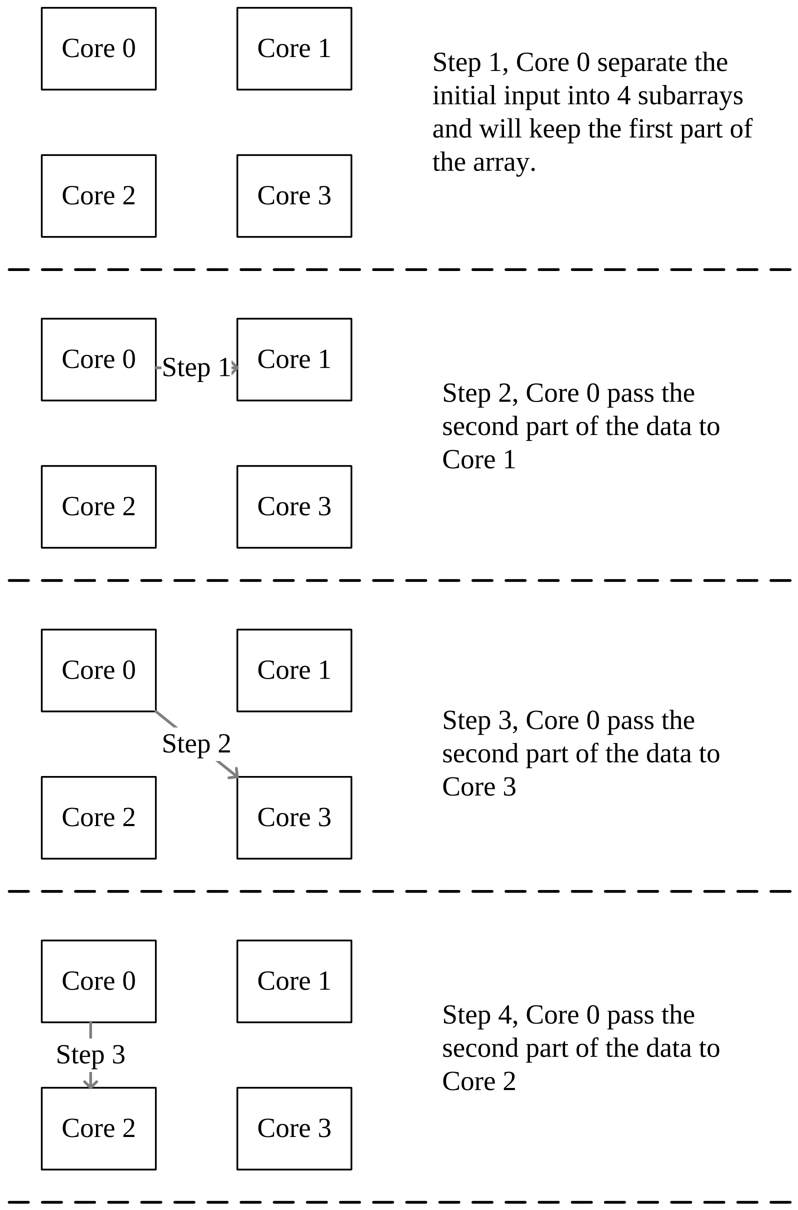

Core 0
Core 1
Core 2
Core 3
Step 1, Core 0 separate the initial input into 4 subarrays and will keep the first part of the array.
Core 0
Step 1
Core 1
Core 2
Core 3
Step 2, Core 0 pass the second part of the data to Core 1
Core 0
Core 1
Step 2
Core 2
Core 3
Step 3, Core 0 pass the second part of the data to Core 3
Core 0
Core 1
Step 3
Core 2
Core 3
Step 4, Core 0 pass the second part of the data to Core 2

Table 5, Spreading the input data to each core directly.

As shown in table 5, it is not an efficient way for spreading data. As all of the data are passing serially, the total time it takes for passing the data would be equivalent to the time it takes for passing the whole input to another core.

Another way to spread data is to pass half of the initial data from Core 0 to Core 2. Then Core 0 and Core 2 pass half of their own data simultaneously to Core 1 and Core 3.

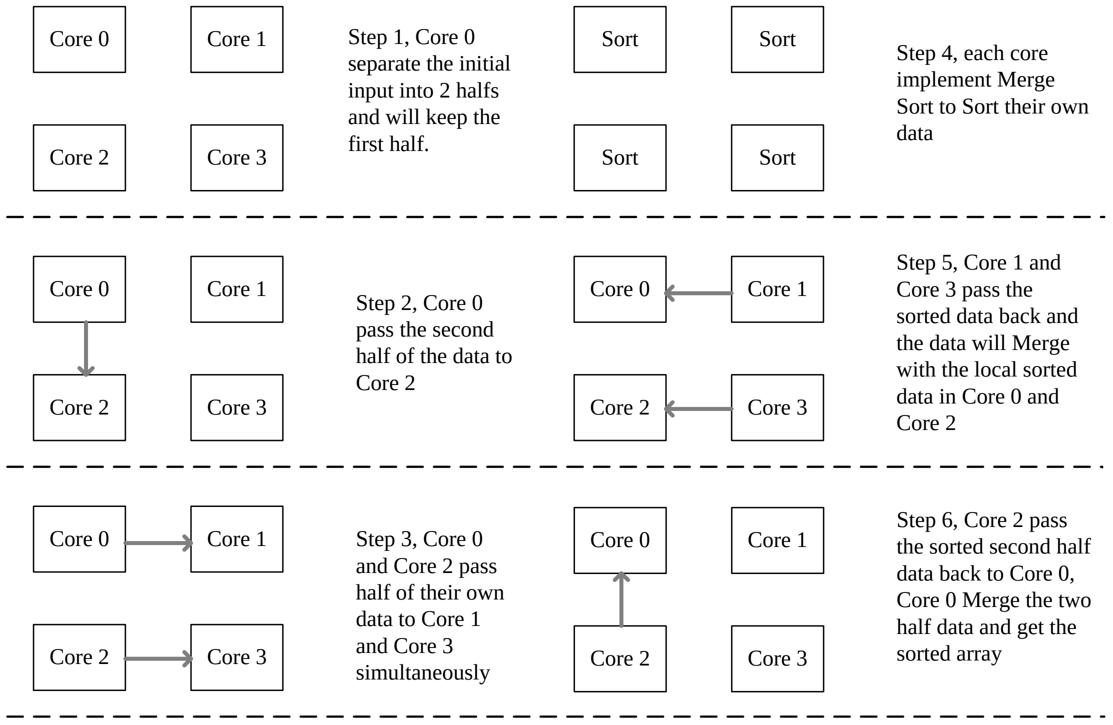

Core 0
Core 1
Core 2
Core 3
Step 1, Core 0 separate the initial input into 2 halfs and will keep the first half.
Sort
Sort
Sort
Sort
Step 4, each core implement Merge Sort to Sort their own data
Core 0
Core 1
Core 2
Core 3
Step 2, Core 0 pass the second half of the data to Core 2
Core 0
Core 1
Core 2
Core 3
Step 5, Core 1 and Core 3 pass the sorted data back and the data will Merge with the local sorted data in Core 0 and Core 2
Core 0
Core 1
Core 2
Core 3
Step 3, Core 0 and Core 2 pass half of their own data to Core 1 and Core 3 simultaneously
Core 0
Core 1
Core 2
Core 3
Step 6, Core 2 pass the sorted second half data back to Core 0, Core 0 Merge the two half data and get the sorted array

Table 6, An improved way to spread the input data.

This method can spread data to each core faster because Core 0 and Core 2 can pass data simultaneously. The total time it takes for passing data would be equivalent to three-quarters of the time it takes for passing the whole input to another core.

3.3.4 Serial implementation of the N-Queens Problem

The N-Queens Problem implemented in this project is intended to found out the number of solutions of the N-Queens Problem on a specific chessboard size N. The number of solutions for the N-Queens Problems on each board size is stationary values which can be found in table 2 page 10. This table can be used to check the correctness of the implementation.

The N-Queens Problem algorithm is implemented by continuing to search for a legal position in which to put a new queen on the chessboard. The flowchart is shown below.

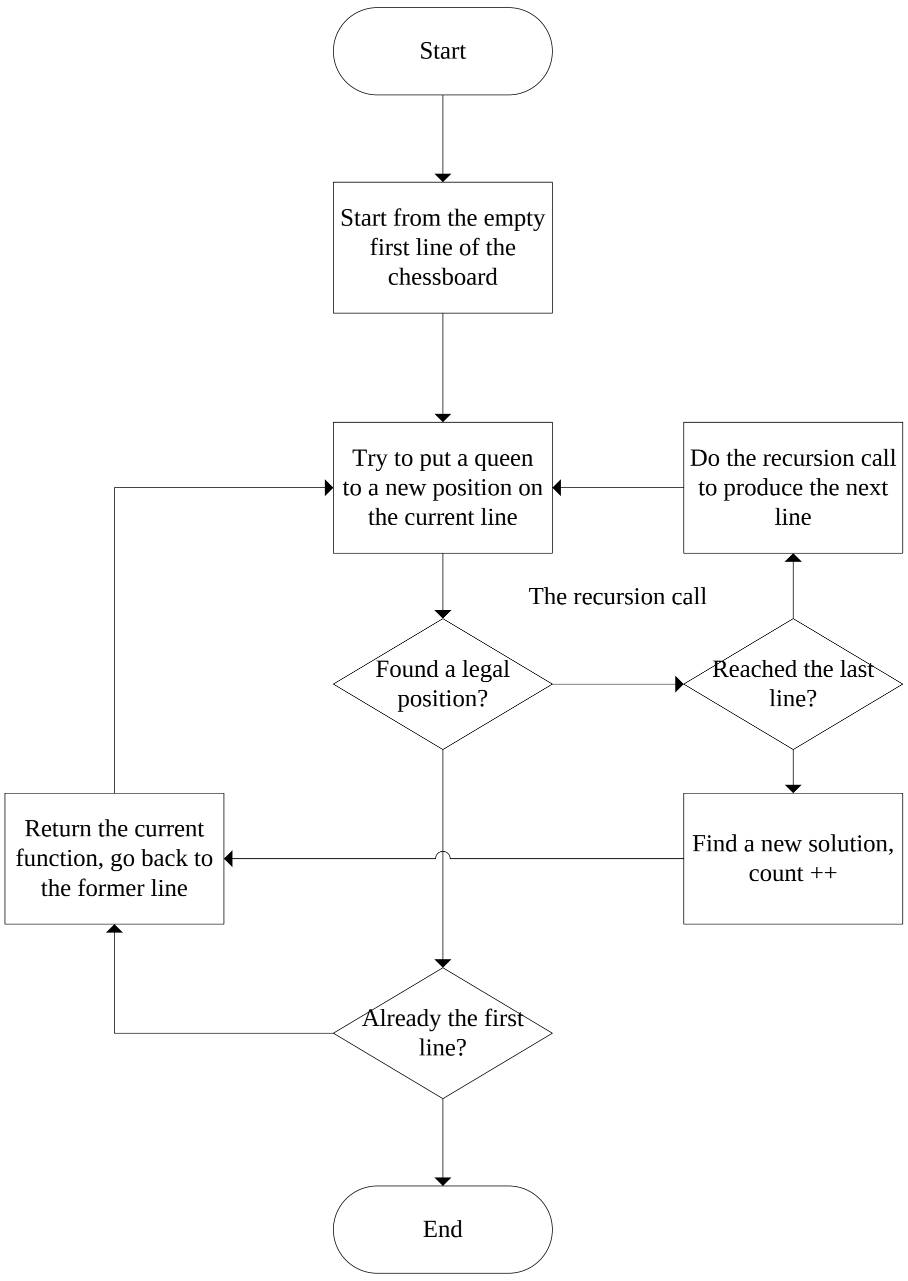

Start
Start from the empty first line of the chessboard
Try to put a queen to a new position on the current line
Do the recursion call to produce the next line
The recursion call
Found a legal position?
Reached the last line?
Return the current function, go back to the former line
Find a new solution, count ++
Already the first line?
End

Figure 15, Flowchart for the N-Queens Problem Algorithm.

The algorithm will keep searching legal positions for a queen line by line until it successfully puts a number of N queens on the N×N chessboard without attacking each other. If the algorithm found a legal position to put the queen on the current line, it will place the queen then keep searching for the next line. If the searching route successfully reached the last line and found a legal position to put the last queen, it means we found a new solution. If there is no legal place to put a queen on a line, it means the current search route failed. When either the current search route failed or we found a new solution, we need to go back to the former line and try to find a new way to place all the queens. After trying all the possibilities, we can know the number of solutions for the N-Queens Problem.

Binary 1 in the bitmaps represents a slot taken or attacked by a queen. The bitmap for a new line can be generated by the three integers ‘left’, ‘right’ and ‘down’. These three integers will be shifted each time we search a new line. More details about the algorithm can be found in reference [13].

3.3.5 Partitioning for the N-Queens Problem

The N-Queens algorithm is about searching for all the possible ways of placing the number of N queens. As each queen will take a whole line, so the algorithm is searching line by line. If the first serial lines of the chessboard have a different placement of queens, the searching route will lead to a totally different group of solutions. This means the N-Queens problem can be partitioned by having a different placement on the first serial lines.

The chessboard bitmap for the next line can be generated by knowing the state of the current line. That means we are able to continue searching later if we stored the state of the current search. To partition the N-Queens Problem, we can first find out the different ways to place queens on the first serial line, and then store the state of all these ways into a two-dimensional array. All these states can be treated as a list of searching routes that can be solved on different cores. The table below shows the number of searching routes found while partitioning with different numbers of lines.

	Number of sub-problems can be partitioned		
Board Size of N-Queens Problem	First Line	First 2 Lines	First 3 Lines
9	9	56	234
10	10	72	364
11	11	90	536
12	12	110	756
13	13	132	-
14	14	156	-
15	15	182	-
16	16	210	-
17	17	240	-
18	18	272	-
19	19	306	-
20	20	342	-
21	21	280	-
22	22	420	-

Table 7, Number of sub-problems can be partitioned.

The states of these sub-problems can be easily gathered by slightly changing the original N-Queens algorithm. In the original N-Queens algorithm, we stop searching when we reach the last line. In the new algorithm, we stop when the search route reaches a point such as the third line, and then pass all the current states to another thread for recording. By doing this, all the possible ways of putting three queens on the first three lines can be stored as a list of search routes.

As the N-Queens Problem will finally be evaluated on the 64-core XMOS device, we must make sure there are enough partitions for the highly parallel device. Each working thread on the device should have at least one partition to make sure the hardware is being fully used. The table below shows the number of working threads when having a different number of cores, with each core running a different number of threads.

	Number of working threads	
Number of cores used	4 threads each core	8 threads each core
2	6	14
4	12	28
8	24	56
16	48	112
32	96	224
64	192	448

Table 8, The total number of working threads.

The parallel N-Queens Problem implementation designed in this project will have to take one thread from each core to do the communication. That explains why the number of total working threads shown in table 8 is less than expected. More details of the implementation can be found in the following sections.

Table 7 and Table 8 can be combined to produce the following figures.

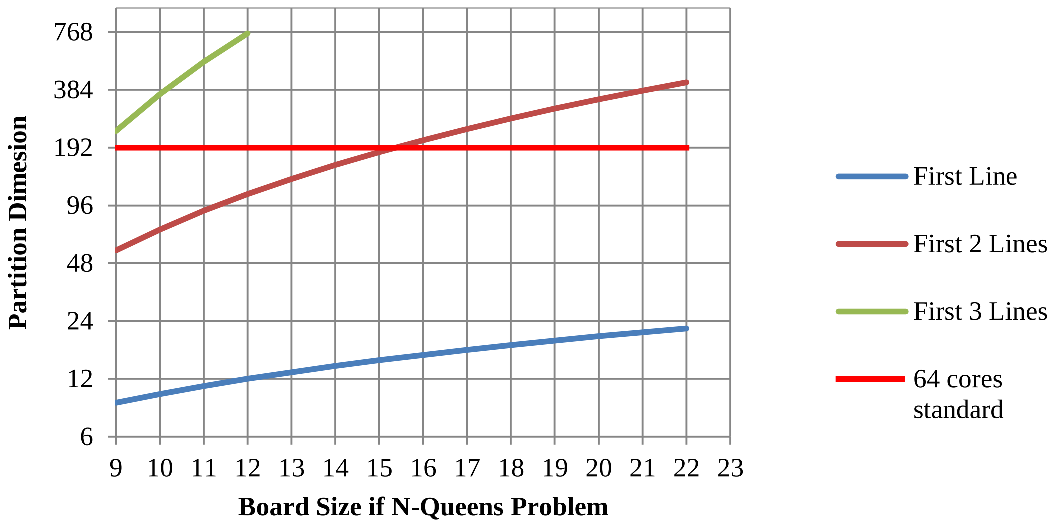


Figure 16, Number of partition match with number of working threads.

Figure 16 shows the standard of how to make full use of the 64-core device, with each core running four threads. To achieve this, the problem should be able to have at least 192 partitions. When the chessboard size is between 9 and 22, partitioning based on the first three lines can always make full use of the 64-core device, and partitioning based on the first line can never make full use of the 64-core device. When partitioning the problem based on the first two lines, the board size should be greater than 16 to make full use of the 64-core device.

However, when partitioning the base on the first three lines, the number of partitions soon becomes too large and the memory on the XMOS device will be used up quickly. In this case, we evaluate two groups of N-Queens implementations.

1. The N-Queens Problem with a Larger Chessboard Sizes. This first group is partitioned based on the first two lines of the board and has board sizes between 16 and 18. Each core used in this implementation has four threads running simultaneously.

2. The N-Queens Problem with a Smaller Chessboard Sizes. This second group is partitioned based on the first three lines of the board and has board sizes between 9 and 12. Each core used in this implementation has four threads running simultaneously.

3.3.6 Designing a more flexible architecture for core connection

3.3.6.1 The ring connection architecture

The Merge Sort implementation described previously uses dedicated core connection architecture with limited flexibility that can hardly fit other kinds of algorithms. Here we introduce a new core connection architecture has more flexibility and is able to fit various algorithms. It also has the potential to handle multi-jobs in a single implementation.

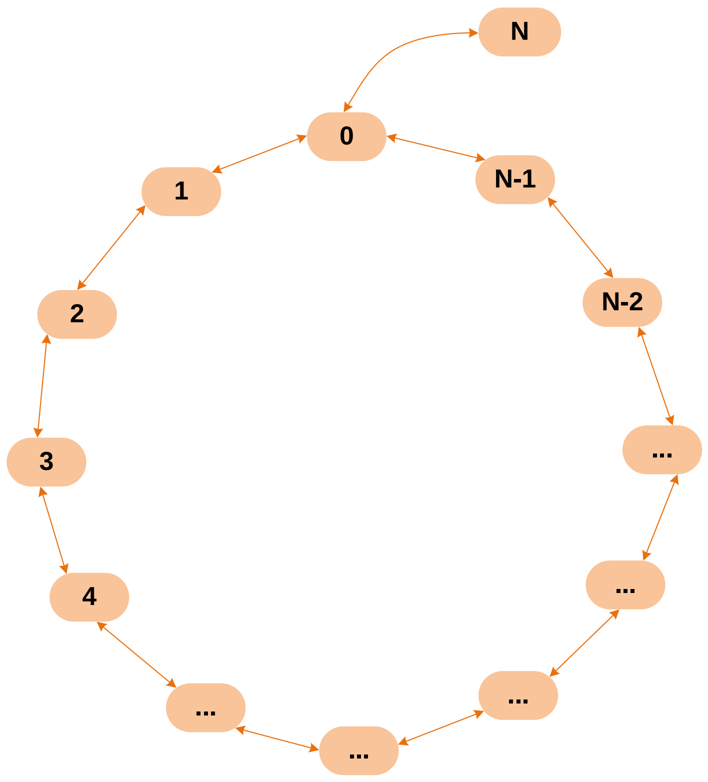


Figure 17, The ring connection architecture.

The figure above shows the core connection designed for the parallel N-Queens Problem implementation. There are a number of N cores connected as a ring. The extra core outside the ring is connected to Core 0 and is used as a timer.

The core connection code for the ring connection architecture is really simple by using the replicator which has been described in reference [14], chapter 3.6. The ring connection can fit any number of cores larger than 3 by simply changing the defining value ‘Core_Num’. This is quit convenient if you compare it to the core connection core for Merge Sort implementation shown in appendix 8.5.

```
streaming chan c[Core_Num];
chan go;
par{
      par (int i=1;i<Core_Num;i++)
      on stdcore[i>=30?(i+1):i] : Node(c[i],c[(i+1)%Core_Num],i);
      on stdcore[0]       : Node0(c[0], c[1], go, 0);
      on stdcore[63]      : Timing(go);
}
```

At the initialization state, Core 0 would act as a master. It will generate the partitions for the N-Queens Problem and send them to the rest of the cores. All the partitions of the N-Queens Problem will pass from one core to another and each core will keep a copy of the partitions. The passing process will be stopped at Core N-1, which is the last core on the ring counting from Core 0. Each core on the ring will start working after finishing their responsibility of passing the copy of partitions to the next core. Core 0 will also join the working group, just as the rest of the cores on the ring.

This ring connection is achieved by taking one thread from each core to do the communication job. The rest of the threads on the core are acting as workers. They will ask for jobs from the communication thread when they are ideal. All the results will be temporarily stored on the communication thread. After all the jobs are complete, all of the results will be gathered to Core 0.

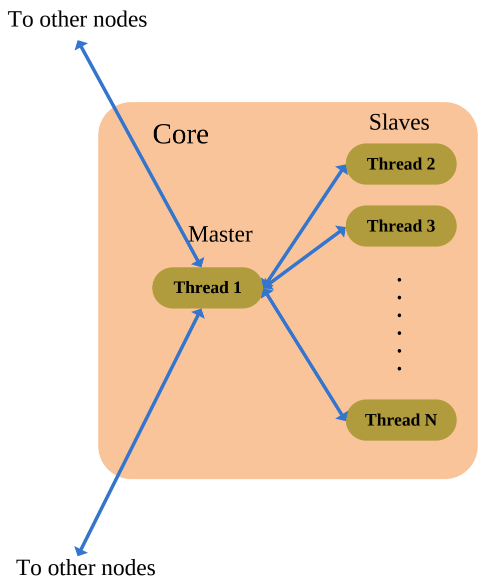


Figure 18, Single node on the ring connection architecture.

In figure 18, Thread 1 acts as a master of other threads on the core. It will assign jobs and store results for other cores. Thread 1 also has channels connected to other cores, which are just the same as the presenting one. All these Thread 1s on different cores form the ring connection and together they will be responsible for the behaviour of the ring connection. Channels between cores use the streaming channels, which can store the passing data on a buffer. [14]

As a hub, Thread 1 will have to receive and send all kinds of data from different channels. To be able to distinguish between the various functionalities, a jump table has been implemented in Thread 1. There are two types of channels connecting to Thread 1: the inner channels connecting between master and slaves, and the outer channels connecting between cores. Different channels will lead to different groups of functions. Also, each bunch of data coming into Thread 1 is started by a notification value. After receiving the notification value, Thread 1 will jump to a specific function and then it will know how to handle the following data.

The codes for the jump table are achieved by using select and case statements, which can be found in appendix 8.6.

Global searching for the ring connection architecture

Global searching technology shall be used while the cores on the ring are sharing a set of data. For the N-Queens implementation on the XMOS device, each core on the ring has a copy of all the inputs, which can also be considered as a list of jobs needing to be done. When a slave thread on a core finishes a piece of work, it will ask for a new job from the master thread. The master thread will choose a piece of work from the job list. To do that, the master thread must know which jobs have already been completed on the entire ring of cores. However, a thread on one core cannot directly access the memory on another core. So the master thread will not be able to know whether a job has already been done by another core. This will cause duplicate work, which will cause some solutions to be counted more than once. It will not only slow down the process but will also make the result of the implementation incorrect. We must figure out a way to let a core know which jobs have been done on the entire ring connection.

To avoid duplicate work, a core needs to search through the whole ring to find out the latest position on the job list when it assigns a new job to a thread. When there is only one core assigning a job on the ring, the situation is simple. The job list is stored in a two-dimensional array. Each core will have a copy of the array, and will also store a number named ‘CP’, which is short for ‘Current Position’, to indicate the current progress on the job list.

Let us assume the searching process starts from a core with core id X. When Core X is going to assign a new job, it will send a notification followed by the value of CP and the core id X to the Core X+1. Core X+1 will realize the ring is checking the latest CP at the moment. It will compare the local CP with the one received from the former core. Core X+1 will pass the same notification, followed by the larger CP and the received core id X to the next core. The passing progress will stop when it is passed back to Core X, as Core X will find that core id X is indicating itself. Core X then gets the latest position on the job list. It then can assign a new piece of work for the pending thread and update the local CP.

Things are more complex in the real implementation where two or more searching processes may run at the same time. As each core has many threads solving a piece of the partition of the original problem, there are many threads doing the job simultaneously on different cores. These threads will be finished at uncertain times and then start waiting for a new assignment. The pending threads may appear on different cores at the same time and new pending threads will keep appearing. A core needs to search on the whole ring to assign a new job, but two or more cores doing the searching at the same time will affect each other. To consider all these possibilities, we can evaluate a ring having six cores as shown in figure 19.

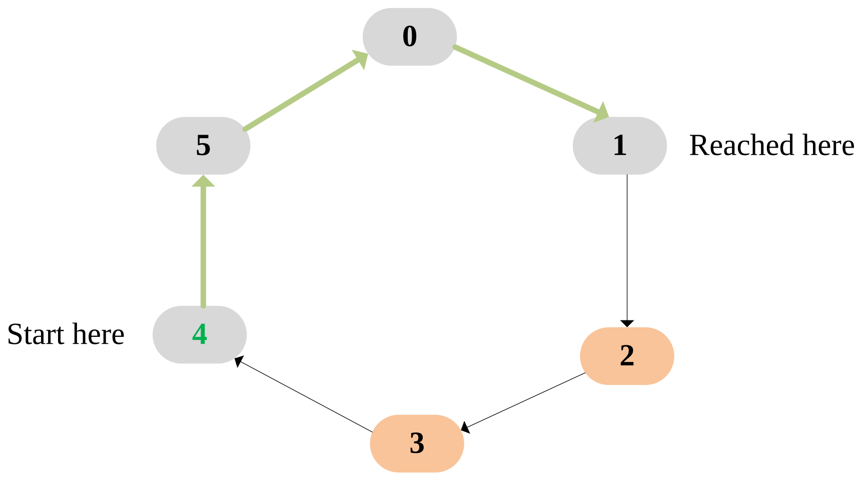


Figure 19, A 6-core global searching model.

In figure 19, Core 4 wants to assign a new job and it starts a searching process on the whole ring. At the present moment, the searching message has passed through Core 5 and Core 0 and reached Core 1. According to this, cores on the ring can be distinguished into two categories:

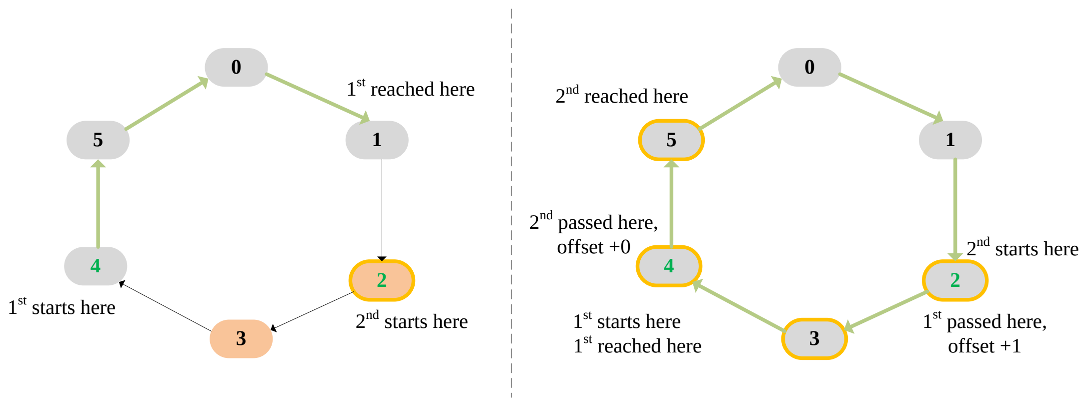
Category A. Cores being reached by the searching process. This category contains cores receiving the searching notification and joining the searching process to find the largest CP on the ring, such as Core 5, Core 0 and Core 1 in figure 19. Cores that start the searching process also belong to this category, such as Core 4 in figure 19.

Category B. Cores have not been reached by the searching process. Cores in this category have not joined any searching process yet. Core 2 and Core 3 in figure 19 are in this category. Cores in Category B will eventually join Category A as the searching process will reach them later.

The second searching process can start on any core in the ring at any moment. The second searching process appearing on a different category of cores can cause different effects and should therefore be treated differently.

1. The second searching process starts on a core in Category B. If this happens, the two searching processes will get the same CP after searching through the ring. This will cause duplicate work and make the final result incorrect. To solve this, an offset mechanism has been introduced. While passing the searching message across the ring, we pass an additional value called Offset, together with the CP and core id. When a core starts a searching process, an indicator variable on this core will be set to 1. When a searching message reaches a core that has started a searching process, a comparison will be made between the local core id and the passing core id. If the passing core id is larger than the local core id, 1 will be added to the Offset. The Offset will be added to the CP when the searching process is finished.

For example, let us assume the second searching process starts at Core 2 at the moment shown in figure 20.

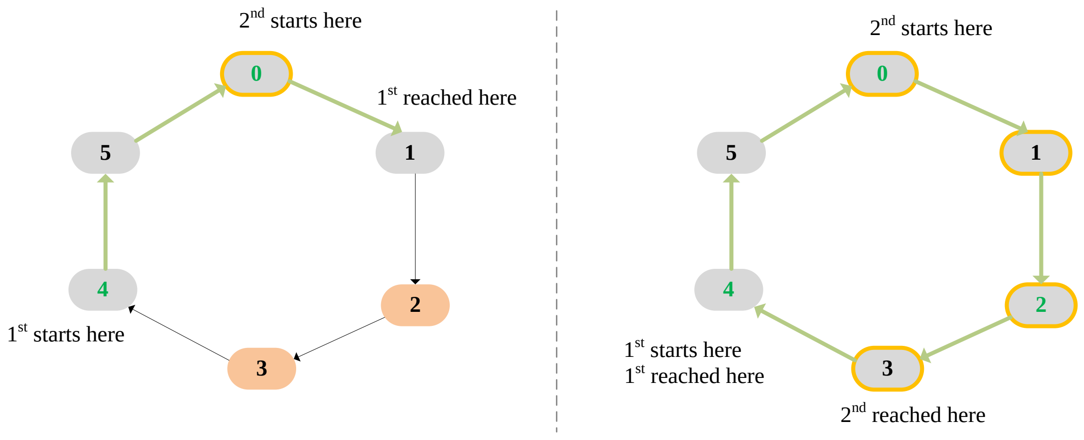


Second searching process start at Core 2 3 searching steps latter

Figure 20, The second searching process starts on a core in Category B.

Figure 20 shows a situation in which a second searching process starts on Core 2, which is in category B. After a certain amount of time, each searching process has passed through three cores. The first searching process reached Core 4, which is where it starts, and the second searching process reached Core 5. During that time, the first searching process has passed Core 2, which starts a searching process and has a core id of 2, which is smaller than the passing core id of 4. A 1 will be added to the Offset of the first searching process. The Offset of the second searching process will not be changed when it passes through Core 4 because the passing core id 2 is smaller than the local core id 4. After the two searching processes finish, both Core 4 and Core 2 got the largest CP. The CPs will be added by different Offsets so there will not be duplicate work.

2. The second searching process starts on a core in Category A. The following example shows what will happen in this situation.



Second searching process start at Core 0 3 searching steps latter

Figure 21, The second searching process starts on a core in Category A.

In this situation, the first searching process is unable to set the Offset value as the second searching process starts at a core that has already been passed. The second searching process is unable to set the Offset value either, as when the second searching process reaches where the first searching process started, the first searching process is already finished. In figure 21, when the first searching process finished, Core 4 will remove the indictor and go back to work. When the second searching process reaches Core 4, it will be unable to realize there is a process starting at this core.

To solve this, we can stop the cores in Category A starting a new searching process. An indicator named Category with an initial value 0 is set on every core in order to distinguish the category of the cores. When a core starts or is reached by a searching process, 1 will be added to the variable Category. When a searching process is finished, the starting core will send a releasing message across the ring to subtract 1 from the indicator Category on each core. When the value of Category on a core is larger than 0, it means this core is in Category A. When the value of Category on a core is still 0, this core is in Category B. Cores in Category A will postpone to start a new searching process until it turns back to Category B.

Chapter 4: Analysis of the Results

4.1 The Results for N-Queens Problem with Larger Chessboard Sizes

Let us first examine the N-Queens Problem with an 18×18 board size, which is a representative situation that clearly shows the power of parallelization. The 18×18 N-Queens Problem has 666090624 numbers of results, which is a huge problem for the XMOS device. The following chart shows the reduction of time for the 18^2 N-Queens implementation with the growth of core numbers. This group of tests are referenced on page 31.

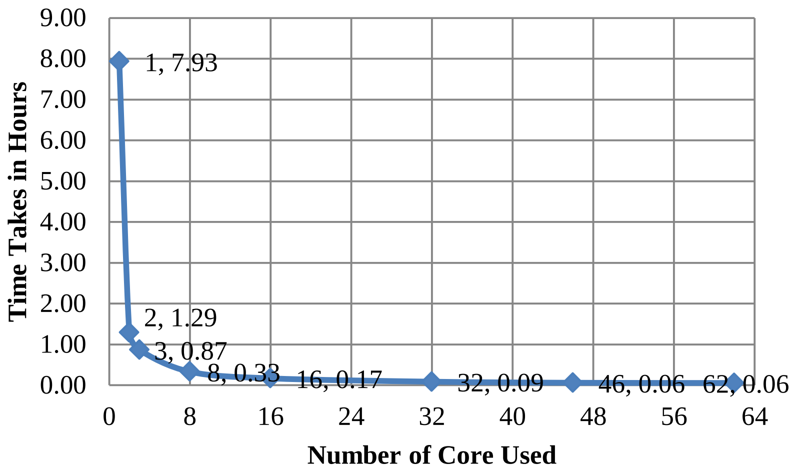


Table 9, Running time for the 18^2 N-Queens Problem implementation.

Table 9 is formed by the set of time taken for solving the 18^2 N-Queens implementation on a different number of cores. The blue line shows the trend of how the running time of the N-Queens implementation is reduced when it is going more parallel. Solving the 18×18 N-Queens Problem in serial, which means on a single thread of the XMOS core, will take approximately 8 hours. When it runs parallel on 8 cores and each core has 4 threads, the running time is reduced to 20 minutes. When it runs on 62 cores, the running time is reduced to 4 minutes. The following chart shows the observed speedup for the parallel N-Queens implementation with a board size of 18^2.

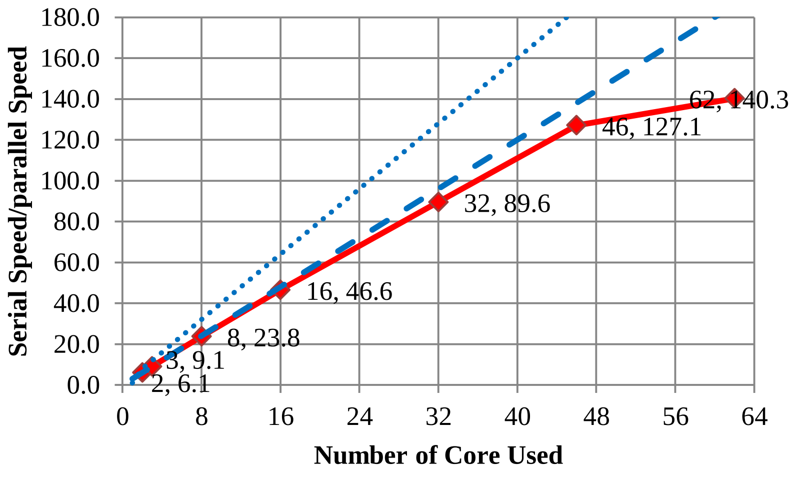


Table 10, Observed speedup for 18×18 N-Queens Problem.

The red line in table 10 is the observed speedup for the parallel 18×18 N-Queens Problem implementation. The observed speedup shows how many times faster the N-Queens implementation runs in parallel than it runs in serial (see chapter 2.5, page 15). As shown in table 10, the observed speedup has a sustained and steady growth before the cores used reached 46. The increased speedup slows down slightly when the core number is larger than 46. This kind of slow down can be seen more clearly when the chessboard size is reduced.

In table 10, the blue dotted line shows the total number of threads used in this implementation, and the blue dashed line shows the number of threads used for solving the partitions of the N-Queens Problem. The N-Queens Problem implementation uses the ring connection architecture as described in chapter 3.3.6, page 32. There are 4 threads used in total in this implementation; one thread is used for communication and the remaining three threads solve the partitions of the N-Queens Problem. The increase in thread number can also be considered as the increase of computational powers.

In table 10, the red line for observed speedup is very close to the blue dashed line when the core number is less than 32. This means the Parallel N-Queens implementation is able to make good use of the increased number of cores. When the implementation goes more parallel, the difference between the blue dashed line and the red line becomes more obvious. That is because there is always some hindering force against parallelization and the hindering force will become even stronger as the implementation goes more parallel.

As shown in table 10, there is always an insurmountable distance between the blue dotted line and the blue dashed line. This is because one thread on each core has to be taken to do the communication job to build the ring connection architecture. Thus the more cores used for the implementation, the more computational power lost. That is the inherent drawback of the ring connection architecture, and it is the trade-off for getting more flexibility.

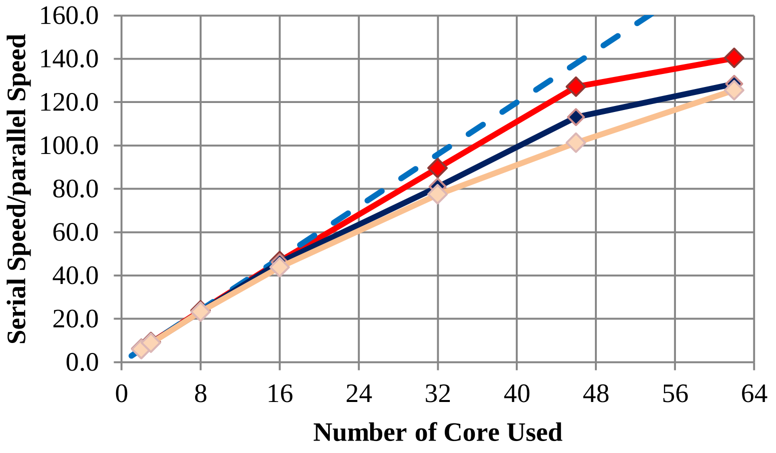


Table 11, Observed speed up for N-Queens Problem with a large board size.

Table 11 shows the observed speedup for the N-Queens Problem with a board size of 16×16, 17×17 and 18×18. The N-Queens Problem with the large board size shares similar characteristics. As shown in table 11, the observed speedup for the N-Queens

Problem with a bigger board size is always greater than the N-Queens problem with smaller board size. The differences become more obvious when the implementation goes more parallel. That is because when the problem size is reduced, the time taken by these hindering forces against parallelization becomes more significant. When the problem size is small enough, the observed speedup may decrease when the implementation goes more parallel.

The entire result for the parallel N-Queens Problem Implementation with various board sizes can be found in Appendices 8.3 and 8.4.

4.2 The Results for the N-Queens Problem with Smaller Chessboard Sizes

This group of tests are referenced on page 31, and refer to the N-Queens Problem with a very small problem size. The table below shows the running time for the parallel N-Queens Problem implementation with a board size from 9×9 to 12×12.

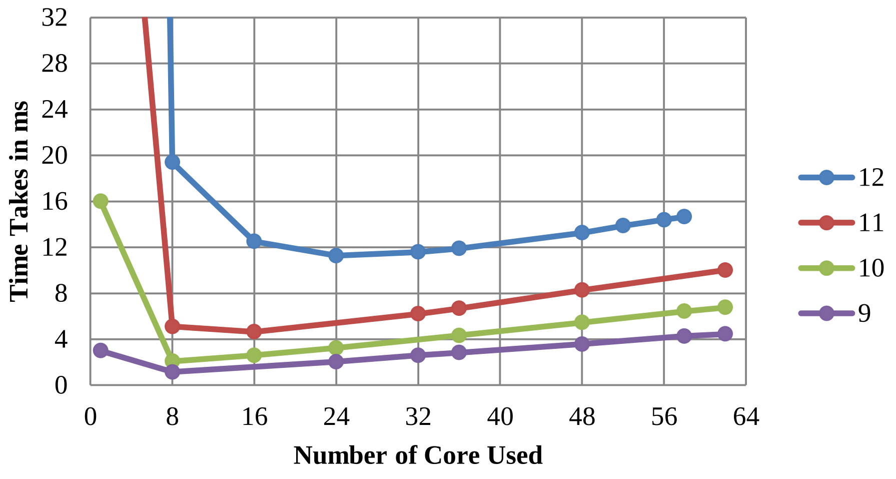


Table 12, Running time for N-Queens Problem with a small board size.

Solving the N-Queens Problem with a board size of 12 and 11 in serial takes 401 milliseconds and 75 milliseconds, this is not shown in table 12. As shown in the table, the implementation takes an even longer time when it is going more parallel. We can see that the smaller the problem size of the implementation, the earlier the slowdown begins. Such as the N-Queens the problem implementation with a board size of 12 will start slowing down when the number of the cores being used is greater than 24. The same implementation but with a board size of 9 will start slowing down when the number of cores being used is greater than 8.

The following table shows the observed speedup for this group of tests. It clearly shows the observed speedup for the implementation decreases when it goes more parallel.

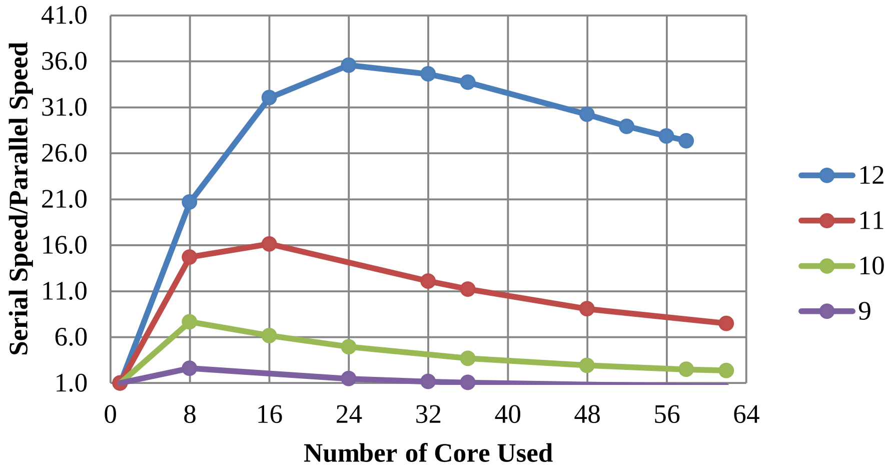


Table 13, Observed speed up for N-Queens Problem with a small board size.

Table 13 demonstrates a situation that the implementations can become even slower when they occupy more cores. This indicates that going more parallel will not always speed things up. We should stop at some point before making an implementation that is too parallelized.

A similar result has been found by other people implementing a different algortihm on a different parallel arhitecture.

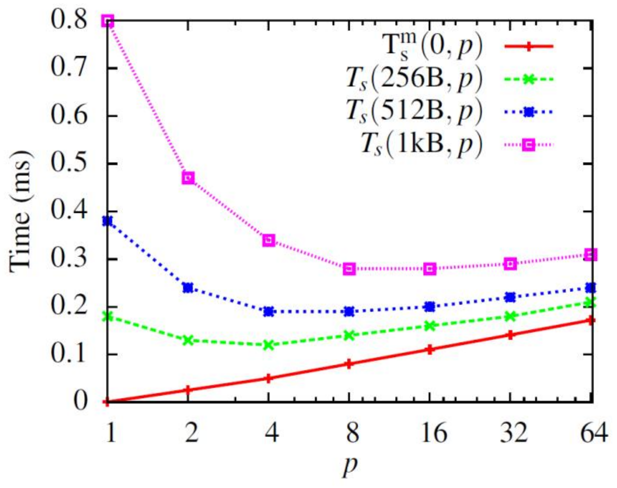


Table 14, a similar result from James Hanlon and Simon J. Hollis [3]

Table 14 shows a result of implementing the Merge Sort on shared memory architecture from James Hanlon and Simon J. Hollis [3]. The input sizes used in table 14 are relatively small input sizes for the shared memory architecture. The Merge Sort in this project is limited by the memory size of the distributed memory architecture and unable to achieve that input size. This shows that the parallel computations are sharing similar characteristics when the problem size is small even on different architectures with different implementations.

The best point to stop for parallelization is different for different implementations. It will depend on the importance of speed and efficiency for a specific situation. Generally, we should stop making an implementation more parallel if the speedup is not impressive. For example, the N-Queens Problem with board size 12: parallelizing it from 16 cores to 24 cores only has a very small speedup but it will occupy 8 more cores. That would be a waste of computational power and energy.

Those implementations with a very small problem size, such as the N-Queens Problem with a board size of 9 or 10, can easily be solved in serial. In general, there is no need to partition them into parallel to generate a small speedup. For parallel computations, these small problems can be treated as single elements. A group of these elements can be solved on different cores simultaneously to take the advantage of parallel computation.

4.3 The Inherent Run Time for Parallel N-Queens Problem Implementation

In order to know exactly what makes the implementation slowdown, we modified the implementation to measure the inherent run time for the parallel N-Queens Problem implementation. Parallel implementation will have to do more work than the serial implementation. The inherent run time for the parallel implementation is the time taken by this extra work. To measure this inherent time, the modified implementation will be no longer solving the partitions of the N-Queens Problem. Instead, the slave threads will return a fake result '1' as soon as they are assigned a new piece of work. In this case, the implementation will still run the procedures which are partitioning the N-Queens Problem, spread the partitions onto each core, assign new pieces of work to the pending threads, and finally collect all the results.

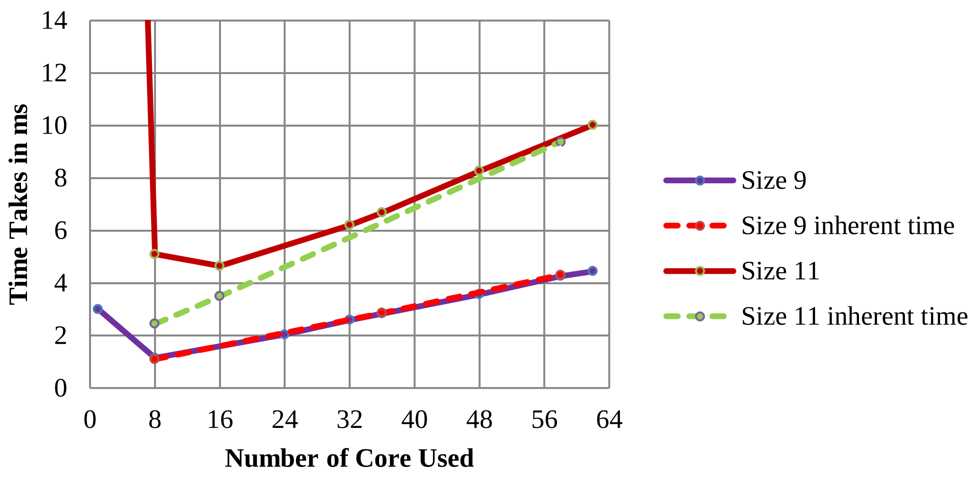


Table 15, The inherent run time for N-Queens Problem implementation.

Table 15 shows a comparison between the total run time and the inherent run time of the parallel N-Queens Problem implementation. The inherent run time for the 11×11 N-Queens Problem, which is the green dashed line, keeps increasing and is nearly matched to the total running time after the core number is greater than 56. The time taken to solve the partitions of the 11×11 N-Queens Problem keeps decreasing and only takes a small part of the total run time. That means when the implementation is running more parallel, the time it takes for solving the partitions has been reduced, but the inherent run time for the implementation has increased, which causes the total running time to increase.

The N-Queens Problem with a board size of 9×9 is a more extreme situation in that almost all the run time is taken by the inherent run time. In the test cases, the inherent run times are even slightly larger than the total running time of the implementation before disabling the partition solving module.

Number of cores	Total time	Inherent time	Inherent time / Total time
8	1.14704	1.10204	96.07%
36	2.83804	2.86304	100.88%
58	4.26704	4.30704	100.94%

Table 16, Time analysis for 9×9 N-Queens Problem implementation.

As shown in table 16, the implementation that is not solving the partitions takes an even longer time than the one solving the partitions. That is possibly because disabling the partition-solving module will cause the pending threads to appear denser. Thus there will be more cores doing the global searching at the same time. The searching process shall be slowed down as each core has to wait until all the other cores have finished their searching. This deviation is very small and it will not be noticed when the problem size is bigger.

The parallel N-Queens Problem implementation on the XMOS device uses the ring connection architecture to connect the cores being used. When the implementation goes more parallel, there would be more cores listed on the ring. Thus the time taken for doing global searching, data spreading and result collecting will be increased. That will make the implementation less efficient when it goes more parallel.

4.4 The Results for Parallel Merge Sort Implementation

The Merge Sort algorithm is an algorithm needed to handle large amounts of data. It was implemented on the 64-core XMOS device and we are going to see how this kind of algorithm can be sped up. In this implementation there are 8 threads run simultaneously on each core. The numbers of cores that can be used have been limited to 7, 15, 31 and 63 by the core connection architecture of this implementation. The length of the input array has been limited to below 6000 by the memory size of the XMOS device. The following table shows the parallel Merge Sort implementation with input sizes of 2k, 4k and 6k. The results for other input sizes can be found in Appendix 8.1.

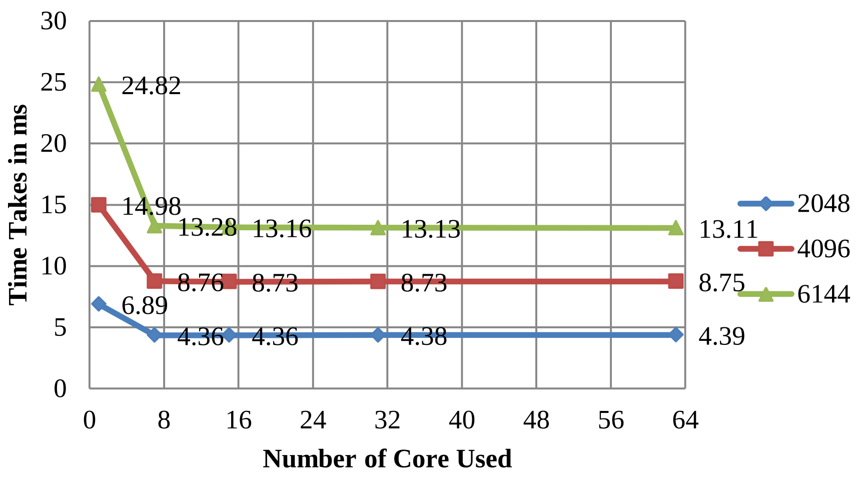


Table 17, The results for parallel Merge Sort implementation.

As shown in table 17, the running times for the Merge Sort implementations both have a small reduction when they are parallelized from a single thread to 8 cores. After that, the implementations can hardly be sped up. The table below shows the observed speedup of the Merge Sort implementation.

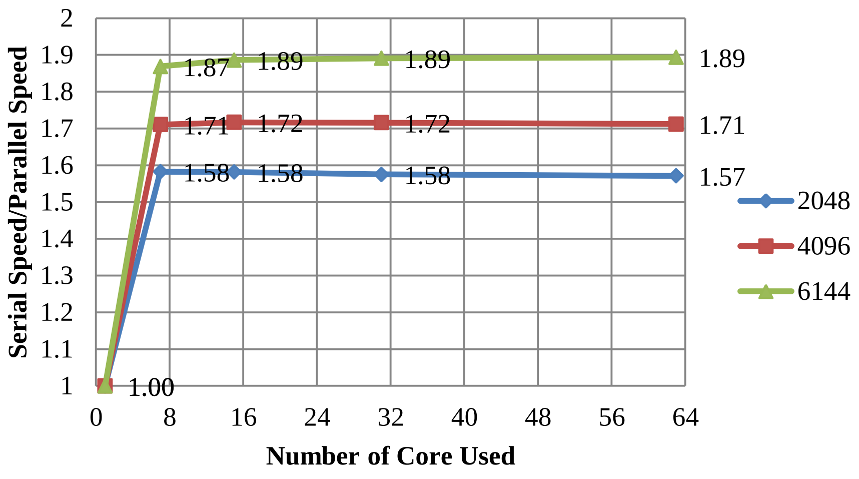


Table 18, Observed speedup for Merge Sort implementation.

According to table 17 and table 18, the Merge Sort algorithm can only have a less than two speedup while it has been parallelized from a single thread to 56 threads on 7 cores. According to reference 14, page 40, the clock rate of a thread will be reduced to keeping a total 400 MHz on a single core. Thus a core having 8 threads only has four times the computational powers than a single thread run at 100 MHz clock rate. Thus, the computational power has increased 28 times from a single thread to 7 cores. However, the observed speedup for this implementation never exceeds 2.

The speedup for the parallel Merge Sort implementation can only change slightly when the core number is greater than 8. As shown in table 17, while having a smaller input size, such as an input array of 2048 and 4096, the implementation can have a slight slowdown when it goes more parallel. On the other hand, having a larger input size such as 6144, the implementation can have a slight speedup when it goes more parallel. That means going more parallel does have some effect on the running time of the implementation. But the implementation must be limited by some shortcomings that stop the implementation from speeding up.

4.5 The analysis for the parallel Merge Sort implementation

In order to figure out what exactly stopped the parallel Merge Sort implementation from speeding up, we divided the running procedure of the implementation into different regions and recorded the running time for each of the regions. The running procedure of the Merge Sort implementation is divided into three regions as shown below.

1. Communication Time. The communication time is purely the time used for transmitting data between cores. It can be recorded by disabling both the sorting modules on the lowest level cores and the module used for separating and merging data on each level of cores. After disabling all these modules, the implementation is actually passing the same amount of random data as the input array to the lowest level cores and then passing this data back to Core 0. The communication time for a Merge Sort implementation with an input size of 6144 is shown in the table below.

Levels for Core Connection	Number of Core Used	Time Takes in ms
5	7	8.328
6	15	8.328
7	31	8.345
8	63	8.438

Table 19, The communication time for Merge Sort with an input size of 6144.

As shown in table 19, the communication time only increases a small amount when passing it to more levels of cores. That is because the array passing to lower level cores will be partitioned into smaller sizes. The largest two partitions, which are the two halves of the initial inputs, need to be passed to the two next level cores. These two partitions will be passing one-by-one by a single thread on Core 0, which will take a very long time. When the data goes to lower level cores, there will be many more numbers of threads passing smaller sizes of partitions simultaneously. That is why passing data to many more cores only takes a very small amount of data.

Table 19 shows the communication time when the implementation are using 7 cores and using 15 cores. That is because in this implementation, the partitions have to be passed to cores in level 6 and then detect whether the cores in level 6 should be used. In future works, the detecting procedure can be moved to cores in level 5 so we do not need to pass data to cores in level 6 if we are not going to use these cores.

2. Times for Separating and Merging Data in level 1 to 4. As shown in figure 12, page 22, the Merge Sort from level 1 to level 4 executes on the first 3 cores. The entire input array will be separated into 16 subarrays at the start of the implementation and the 16 subarrays will be merging to an entire array at the end of the implementation. This work is being done by a single thread on Core 0 and the first three threads on Core 1 and Core 2, which will take a significant amount of time. With a different number of cores used in the Merge Sort implementation, the process in the first three cores remained the same. The Merge Sort implementation with an input size of 6144 on a different number of cores will take a constant 4.37 milliseconds for separating and merging data in level 1 to 4.

3. Time for Lower Level Sort. The lower level sort means the Merge Sort implementation can be executed on cores from level 5 to level 8, as shown in figure 12, page 22. This time can be calculated by subtracting the execution time for higher level cores from the entire execution time. The execution time for higher level cores is the execution time taken by cores from level 1 to level 4. The table below shows the calculation of the time for lower level sort. All the times shown in table 20 are given in milliseconds.

Levels for Core Connection	Number of Core Used	Entire Execution Time	Time for Higher Level	Time for Lowe Level
5	7	13.282	12.135	1.147
6	15	13.16	12.135	1.025
7	31	13.127	12.135	0.992
8	63	13.108	12.135	0.973

Table 20, The calculation for the time of lower level sort.

By adding the three kinds of time together we can get the approximate execution time for the Merge Sort implementation. The table below shows the accumulation analysis of the execution time of the Merge Sort implementation. The communication time used here is only the communication time for cores in level 1 to level 5. That is because the communication time for higher level sort has already been included in the time for lower level sort.

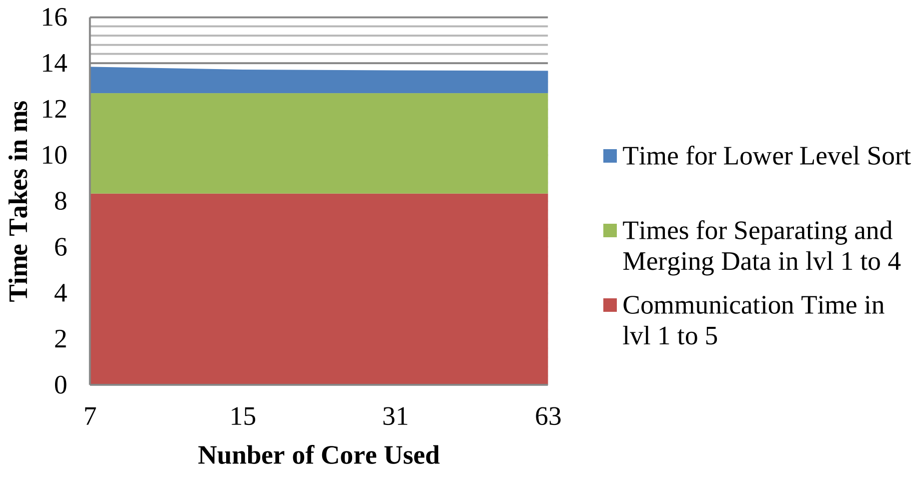


Table 21, The accumulation analysis of the Merge Sort implementation.

It can be seen from table 21 that most of the execution time for the Merge Sort implementation has been taken by the cores in higher levels, such as Core 0, Core 1 and Core 2, shown in figure 12 on page 22. Adding a new level of cores can only reduce the execution time for the lower level sort. The constant execution time on higher level cores still remains and that is why the Merge Sort implementation stopped speeding up while going more parallel.

In conclusion, the parallel Merge Sort implementation shows the following two factors will go against parallelization.

1. Data Communication. Parallel implementation may require more data communication between cores than the serial implementation. An algorithm that requires a large amount of communication will become less efficient when it is being parallelized.

2. Dependency. In parallel implementation, the execution on one core may rely on the result from another core. For Merge Sort implementations the lower level cores need to wait for the higher level cores to pass the data to them before they can be executed. The higher level cores need to wait for the lower level cores passing the sorted subarrays back before they can merge the subarrays together. Thus only one level of cores can be executed at a time.

4.6 Result for Optimized parallel Merge Sort on a 4-core device

The table below shows the results of a Merge Sort implementation optimized on the XMOS XC-1 development kit, which has 4 cores. This implementation tries to optimize the data flow of the Merge Sort implementation to obtain better speedup.

Threads used on each core	Number of cores	Threads used in total	Time takes in ms	Observed speedup
1	1	1	6.894	1
4	1	4	3.432	2.01
4	2	8	2.958	2.33
4	4	16	2.816	2.45

Table 22, Result for the optimized Merge Sort implementation.

As shown in table 22, the optimized Merge Sort implementation on the 4-core device has achieved a higher speedup than the original implementation on the 64-core device.

This example demonstrate that an algorithm have a frequent data communication, as described in chapter 2.5.1, can hardly achieve an impressive speed up during parallelization. The optimized Merge Sort implementation is using an improved method passing data through cores as shown in table 6 page 27. Merge Sort as an algorithm which has a frequent data communication, optimizing the data flow can significantly reduce its execution time. However, because of the unavoidable large amount of data communication, the speedup is still not very impressive.

Chapter 5: Critical Evaluation

This project has achieved most of its objectives. This chapter analyses how those objectives were achieved, why the choices were made and explains the conclusions and principles found in this project.

5.1 The parallel implementations

This project chose to implement fewer algorithms than we investigated during the background reading. By doing this, we had more time to focus on each implementation and were able to do deep evaluations on them.

The implementations in this project are sufficient even if it only used a limited number of serial algorithms, which are Merge Sort and the N-Queens Problem. That is because a serial algorithm is only a bottom level element of parallel implementation and it can form different kinds of implementations by having different data flow, core connection and input sizes. The Merge Sort has been implemented in two implementations: one is able to scale on the 64-core XMOS device and the other one is highly optimized on the 4-core XMOS device. The parallel N-Queens Problem implementation shows the characteristics of embarrassing parallelization when it has a large input size and it demonstrates how the communication slows down the implementation when it has a small input size. Also, the N-Queens implementation uses a more flexible architecture, which contrasts sharply with the Merge Sort implementation.

The comparison between the architecture of the Merge Sort implementation and the N-Queens Problem implementation shows that we need to make a compromise between efficiency and flexibility. The new ring connection architecture obtained more flexibility for the N-Queens Implementation. However, it will take one thread from each core to form the ring connection architecture, which will cause a constant loss of computational power. On the other hand, a parallel implementation can be optimized for a particular device to obtain more efficiency, such as the Merge Sort implementation optimized on the 4-core device. However, the highly-optimized implementation can hardly be scaled up.

5.2 Analysis of the results

The implementations designed in this project have shown a variety of situations in parallel computation. Analysing these situations can help us to realize the problems that shall be faced on our path to parallelization and to detect the forces that hinder parallelization.

The result of parallel N-Queens implementation with a large board size demonstrated the power of parallelization. It shows an ideal situation that the implementation got a very impressive speedup after it goes parallel. This is an optimistic result that is expected by many engineers.

The result of the parallel N-Queens implementation with a small board size shows another situation in which an implementation could be slowed down while going more parallel. That is because the parallel implementation will have to do more work than a serial implementation. When that extra work consumes more time than the time saved by using more cores, the implementation is over-parallelized.

The parallel Merge Sort implementation can only have a small speedup when it runs in parallel on 8 cores, and it cannot get a further speedup by using more cores. That is because the parallel Merge Sort implementation will have a larger amount of data communications. Also, in the Merge Sort implementation, cores in different levels cannot run simultaneously because of the feature of the Merge Sort algorithm. That makes the run time of the Merge Sort implementation limited by the execution time of higher level cores.

5.3 The hindering forces against parallelization

As we can see from these implementations, there are many hindering forces that will go against parallelization. Some hindering forces are caused by the extra work that has to be done by parallel implementation, such as partitioning the problem, data communication between cores, forming the result of partitions together, and so on. Some are caused by features of the algorithms, such as in the Merge Sort implementation, only one level of cores can run at a time. Some of the hindering forces can be overcome, but some are unavoidable. We must keep these hindering forces in mind and try to avoid them or make them less effective when writing a parallel code.

5.4 Optimize and re-implement

After knowing exactly what slows down the execution of these implementations, some of the implementations have been optimized and re-implemented. In the Merge Sort implementation, the functions run on higher level cores have been optimized. By making the merging procedure on Core 1 and Core 32 run parallel on each core, the execution time for the merging process on Core 1 and Core 32 has been reduced from 2.89 ms to 1.73 ms with an input size of 6144. As the improvement is not impressive, we highly optimized the Merge Sort implementation on the 4-core device. The optimized implementation has achieved a higher speedup than the original implementation by using a fewer number of cores. However, the optimization process has broken the Merge Sort algorithm and makes it a totally different code than the original implementation. The code size of the optimized implementation, which was executed on the 4-core device, has exceeded the code size of the original implementation, which was executed on the 64-core device. The optimization makes the implementation lack flexibility and it would become extremely complex if we scale the optimized implementation onto the 64-core device.

Chapter 6: Suggestions and Further Work

There are still some further tests that can be done based on the current implementations, and the current implementations also have the potential to be improved. This chapter gives some suggestions and further work could be done in the future.

1. Further tests for the implementation of N-Queens Problem.

The N-Queens Problem has been tested with each core having 4 cores running simultaneously. The results for those tests with a large board sizes have shown that a quarter of the computation has been lost. That reflects that there is one thread being taken from each core to form the ring connection architecture.

In further work, we can have more tests of the N-Queens Problem implementation but have 8 threads running simultaneously. An improvement is expected to be seen as only one-eighth of the computational power is taken to form the ring connection architecture in this instance. Also, because less computational power is used to do the communication, the hindering forces caused by data communication would become more significant.

2. Implement multiple algorithms onto the ring connection architecture.

The ring connection architecture introduced in chapter 3.3.6, page 32 is a good compromise between flexibility and efficiency, which is very suitable for parallel computation. This ring connection architecture has the potential to handle multiple algorithms simultaneously.

To achieve that, we can have partitions from different algorithms on the same job list and set a notification bit to indicate the origin of each partition. Then the slave threads can recognize the partitions from different algorithms and handle them separately.

Chapter 7: Bibliography

[1] Rusu, S.; Tam, S.; Muljono, H.; Stinson, J.; Ayers, D.; Chang, J.; Varada, R.; Ratta, M.; Kottapalli, S.; Vora, S.; , "A 45 nm 8-Core Enterprise Xeon¯ Processor," Solid-State Circuits, IEEE Journal of , vol.45, no.1, pp.7-14, Jan. 2010
doi: 10.1109/JSSC.2009.2034076

[2] Steve Plimpton, Sudip Dosanjh, Randy Krall, and Y Krall. Is simd enough for scienti_c and engineering applications on massively parallel computers?, Compcon Spring '92. Thirty-Seventh IEEE Computer Society International Conference, Digest of Papers. , vol., no., pp.95-102, 24-28 Feb 1992 URL:
http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=186694&isnumber=4763

[3] Hanlon, JW & Hollis, SJ. 'Fast Distributed Process Creation with the XMOS XS1 Architecture', Communicating Process Architectures, Communicating Process Architectures 2011, 33, (pp. 195-207), 2011. ISSN: 1383-7575, ISBN: 9781607507734

[4] D. MacMillen, R. Camposano, D. Hill, and T.W. Williams. An industrial view of electronic design automation. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 19(12):1428-1448, dec 2000. URL:
http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=898825&isnumber=19457

[5] Michael J. Flynn. Some computer organizations and their Effectiveness. Computers, IEEE Transactions on, C-21(9):948-960, sept. 1972. URL:

http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5009071&isnumber=5009065

[6] Blaise Barney. Introduction to Parallel Computing. Lawrence Livermore National Laboratory. URL: https://computing.llnl.gov/tutorials/parallel_comp/

[7] David, May, Hend, Muller, XCORE XS1 Architecture Tutorial", Version 1.1, 2009 URL: http://www.xmos.com/published/xcore-xs1-architecture-tutorial

[8] XMOS company, "XMOS XS1 Product Family" URL:
http://www.xmos.com/published/xs1-family-product-brief

[9] Thomas H. Cormen; Charles E. Leiserson; Ronald L. Rivest; Clifford Stein. Introduction to Algorithms, Third Edition. The MIT Press, 3 edition, September 2009.

[10] John L. Gustafson. Reevaluating amdahl's law. Communications of the ACM, 31:532-533, 1988

[11] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In Proceedings of the April 18-20, 1967, spring joint computer conference, AFIPS '67 (Spring), pages 483{485, New York, NY, USA, 1967. ACM.

[12] Michael J. Quinn, Parallel Programming in C With Mpi and Openmp. Mcgraw

Hill Higher Education, September 2003.

[13] E-mail: takaker@ic-net.or.jp, http://www.ic-net.or.jp/home/takaken/e/queen/

[14] Douglas Watt, Programming XC on XMOS Devices, 2009, URL: http://www.xmos.com/system/files/xcuser_en.pdf

[15] Bo Bernhardsson. 1991. Explicit solutions to the N-queens problem for all N. SIGART Bull. 2, 2 (February 1991), 7-. DOI=10.1145/122319.122322 http://doi.acm.org/10.1145/122319.122322

[16] Hanlon, J.; , "The XMOS XK-XMP-64 development board," Networks on Chip (NoCS), 2011 Fifth IEEE/ACM International Symposium on , vol., no., pp.255-256, 1-4 May 2011
URL:
http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5948572&isnumber=5948548

Chapter 8: Appendix

8.1 Results for Merge Sort implementation on XK-XMP-64

Input Size	Depth of Core	Threads Each Core	Number of Cores	Total Threads	Time ms
1024	serial	1	1	1	3.148
	5	8	7	32	2.181
	6	8	15	64	2.2
	7	8	31	128	2.215
	8	8	63	256	2.224
2048	serial	1	1	1	6.894
	5	8	7	32	4.356
	6	8	15	64	4.359
	7	8	31	128	4.376
	8	8	63	256	4.387
3027	serial	1	1	1	11.357
	5	8	7	32	6.505
	6	8	15	64	6.464
	7	8	31	128	6.475
	8	8	63	256	6.474
4096	serial	1	1	1	14.983
	5	8	7	32	8.759
	6	8	15	64	8.727
	7	8	31	128	8.732
	8	8	63	256	8.751
5120	serial	1	1	1	21.138
	5	8	7	32	11.099
	6	8	15	64	10.984
	7	8	31	128	10.946
	8	8	63	256	10.947
6144	serial	1	1	1	24.824
	5	8	7	32	13.282
	6	8	15	64	13.16
	7	8	31	128	13.127
	8	8	63	256	13.108

8.2 Results for Merge Sort Optimized on XC-1A

Input Size	Threads Each Core	Number of Cores	Total Threads	Time ms
1024	4	1	4	1.636
		2	8	1.438
		4	16	1.391
	8	1	8	1.756
		2	16	1.515
		4	32	1.437
2048	4	1	4	3.432
		2	8	2.958
		4	16	2.816
	8	1	8	3.678
		2	16	3.099
		4	32	2.908
3027	4	1	4	5.397
		2	8	4.528
		4	16	4.24
	8	2	16	4.72
		4	32	4.372

8.3 Results for Nqueens Problem with Larger Board Sizes

Board Size	Number of Core Used	Seconds	Board Size	Number of Core Used	Seconds
13	1	2.19	16	1	527.458
	2	0.37		2	88.42
	3	0.249		3	59.401
	8	0.099		8	22.655
	16	0.052		16	12.038
	32	0.034		32	6.814
14	1	12.82		46	5.207
	2	2.164		62	4.202
	3	1.457	17	1	3757.849
	8	0.571		2	629.078
	16	0.309		3	421.198
	32	0.175		8	160.658
15	1	80.198		16	81.57
	2	13.438		32	46.395
	3	9.082		46	33.235
	8	3.522		62	29.249
	16	1.805	18	2	4650.37
	32	0.957		3	3121.334
	46	0.852		8	1199.163
	60	0.594		16	613.01
				32	318.815
				46	224.639
				62	203.515

8.4 Results for Nqueens Problem with Smaller Board Size

Board Size	Core Used	Time in ms	Board Size	Core Used	Time in ms
12	1	401.000	10	1	16.000
	8	19.392		8	2.088
	16	12.514		16	2.595
	24	11.271		24	3.236
	32	11.585		36	4.333
	36	11.894		48	5.460
	48	13.268		58	6.435
	52	13.869		62	6.764
	56	14.394	9	1	3.000
	58	14.657		8	1.147
11	1	75.000		24	2.042
	8	5.102		32	2.594
	16	4.648		36	2.838
	32	6.206		48	3.571
	36	6.681		58	4.267
	48	8.268		62	4.449
	62	10.017			

8.5 Core Connection Codes Comparison between Merge Sort and Nqueens Problem

2. Core connection for Nqueens Problem:

```
streaming chan c[Core_Num];
chan go;
par{
        par (int i=1;i<Core_Num;i++)
        on stdcore[i>=30?(i+1):i] : Node(c[i],c[(i+1)%Core_Num],i);
        on stdcore[0]       : Node0(c[0], c[1], go, 0);
        on stdcore[63]      : Timing(go);
}
```

1. Core connection for Merge Sort:

```
int main(void)
{
        chan c[Core_Num*4];
        par
        {
                on stdcore[0]       : mainCore0(c[1*4],c[1*4+1],c[1*4+2],c[1*4+3],c[32*4],c[32*4+1],c[32*4+2],c[32*4+3],0);
                on stdcore[1]       : MergeFunctionLv0(c[1*4],c[1*4+1],c[1*4+2],c[1*4+3],c[2*4],c[2*4+1],c[2*4+2],c[2*4+3],c[17*4],c[17*4+1],c[17*4+2],c[17*4+3],1);
                on stdcore[2]       : MergeFunction(c[2*4],c[2*4+1],c[2*4+2],c[2*4+3],c[3*4],c[3*4+1],c[3*4+2],c[3*4+3],c[10*4],c[10*4+1],c[10*4+2],c[10*4+3],2);
                on stdcore[3]       : MergeFunction(c[3*4],c[3*4+1],c[3*4+2],c[3*4+3],c[4*4],c[4*4+1],c[4*4+2],c[4*4+3],c[7*4],c[7*4+1],c[7*4+2],c[7*4+3],3);
                on stdcore[4]       : MergeFunction(c[4*4],c[4*4+1],c[4*4+2],c[4*4+3],c[5*4],c[5*4+1],c[5*4+2],c[5*4+3],c[6*4],c[6*4+1],c[6*4+2],c[6*4+3],4);
                on stdcore[5]       : MergeFunction(c[5*4],c[5*4+1],c[5*4+2],c[5*4+3],null,null,null,null,null,null,null,null,5);
                on stdcore[6]       : MergeFunction(c[6*4],c[6*4+1],c[6*4+2],c[6*4+3],null,null,null,null,null,null,null,null,6);
                on stdcore[7]       : MergeFunction(c[7*4],c[7*4+1],c[7*4+2],c[7*4+3],c[8*4],c[8*4+1],c[8*4+2],c[8*4+3],c[9*4],c[9*4+1],c[9*4+2],c[9*4+3],7);
                on stdcore[8]       : MergeFunction(c[8*4],c[8*4+1],c[8*4+2],c[8*4+3],null,null,null,null,null,null,null,null,8);
                on stdcore[9]       : MergeFunction(c[9*4],c[9*4+1],c[9*4+2],c[9*4+3],null,null,null,null,null,null,null,null,9);
                on stdcore[10]      : MergeFunction(c[10*4],c[10*4+1],c[10*4+2],c[10*4+3],c[11*4],c[11*4+1],c[11*4+2],c[11*4+3],c[14*4],c[14*4+1],c[14*4+2],c[14*4+3],10);
                on stdcore[11]      : MergeFunction(c[11*4],c[11*4+1],c[11*4+2],c[11*4+3],c[12*4],c[12*4+1],c[12*4+2],c[12*4+3],c[13*4],c[13*4+1],c[13*4+2],c[13*4+3],11);
                on stdcore[12]      : MergeFunction(c[12*4],c[12*4+1],c[12*4+2],c[12*4+3],null,null,null,null,null,null,null,null,12);
                on stdcore[13]      : MergeFunction(c[13*4],c[13*4+1],c[13*4+2],c[13*4+3],null,null,null,null,null,null,null,null,13);
                on stdcore[14]      : MergeFunction(c[14*4],c[14*4+1],c[14*4+2],c[14*4+3],c[15*4],c[15*4+1],c[15*4+2],c[15*4+3],c[16*4],c[16*4+1],c[16*4+2],c[16*4+3],14);
                on stdcore[15]      : MergeFunction(c[15*4],c[15*4+1],c[15*4+2],c[15*4+3],null,null,null,null,null,null,null,null,15);
                on stdcore[16]      : MergeFunction(c[16*4],c[16*4+1],c[16*4+2],c[16*4+3],null,null,null,null,null,null,null,null,16);
                on stdcore[17]      : MergeFunction(c[17*4],c[17*4+1],c[17*4+2],c[17*4+3],c[18*4],c[18*4+1],c[18*4+2],c[18*4+3],c[25*4],c[25*4+1],c[25*4+2],c[25*4+3],17);
                on stdcore[18]      : MergeFunction(c[18*4],c[18*4+1],c[18*4+2],c[18*4+3],c[19*4],c[19*4+1],c[19*4+2],c[19*4+3],c[22*4],c[22*4+1],c[22*4+2],c[22*4+3],18);
                on stdcore[19]      : MergeFunction(c[19*4],c[19*4+1],c[19*4+2],c[19*4+3],c[20*4],c[20*4+1],c[20*4+2],c[20*4+3],c[21*4],c[21*4+1],c[21*4+2],c[21*4+3],19);
                on stdcore[20]      : MergeFunction(c[20*4],c[20*4+1],c[20*4+2],c[20*4+3],null,null,null,null,null,null,null,null,20);
                on stdcore[21]      : MergeFunction(c[21*4],c[21*4+1],c[21*4+2],c[21*4+3],null,null,null,null,null,null,null,null,21);
                on stdcore[22]      : MergeFunction(c[22*4],c[22*4+1],c[22*4+2],c[22*4+3],c[23*4],c[23*4+1],c[23*4+2],c[23*4+3],c[24*4],c[24*4+1],c[24*4+2],c[24*4+3],22);
                on stdcore[23]      : MergeFunction(c[23*4],c[23*4+1],c[23*4+2],c[23*4+3],null,null,null,null,null,null,null,null,23);
                on stdcore[24]      : MergeFunction(c[24*4],c[24*4+1],c[24*4+2],c[24*4+3],null,null,null,null,null,null,null,null,24);
                on stdcore[25]      : MergeFunction(c[25*4],c[25*4+1],c[25*4+2],c[25*4+3],c[26*4],c[26*4+1],c[26*4+2],c[26*4+3],c[29*4],c[29*4+1],c[29*4+2],c[29*4+3],25);
                on stdcore[26]      : MergeFunction(c[26*4],c[26*4+1],c[26*4+2],c[26*4+3],c[27*4],c[27*4+1],c[27*4+2],c[27*4+3],c[28*4],c[28*4+1],c[28*4+2],c[28*4+3],26);
                on stdcore[27]      : MergeFunction(c[27*4],c[27*4+1],c[27*4+2],c[27*4+3],null,null,null,null,null,null,null,null,27);
                on stdcore[28]      : MergeFunction(c[28*4],c[28*4+1],c[28*4+2],c[28*4+3],null,null,null,null,null,null,null,null,28);
                on stdcore[29]      : MergeFunction(c[29*4],c[29*4+1],c[29*4+2],c[29*4+3],c[30*4],c[30*4+1],c[30*4+2],c[30*4+3],c[31*4],c[31*4+1],c[31*4+2],c[31*4+3],29);
                on stdcore[31]      : MergeFunction(c[30*4],c[30*4+1],c[30*4+2],c[30*4+3],null,null,null,null,null,null,null,null,30);
                on stdcore[32]      : MergeFunction(c[31*4],c[31*4+1],c[31*4+2],c[31*4+3],null,null,null,null,null,null,null,null,31);
                on stdcore[33]      : MergeFunctionLv0(c[32*4],c[32*4+1],c[32*4+2],c[32*4+3],c[33*4],c[33*4+1],c[33*4+2],c[33*4+3],c[48*4],c[48*4+1],c[48*4+2],c[48*4+3],32);
                on stdcore[34]      : MergeFunction(c[33*4],c[33*4+1],c[33*4+2],c[33*4+3],c[34*4],c[34*4+1],c[34*4+2],c[34*4+3],c[41*4],c[41*4+1],c[41*4+2],c[41*4+3],33);
                on stdcore[35]      : MergeFunction(c[34*4],c[34*4+1],c[34*4+2],c[34*4+3],c[35*4],c[35*4+1],c[35*4+2],c[35*4+3],c[38*4],c[38*4+1],c[38*4+2],c[38*4+3],34);
                on stdcore[36]      : MergeFunction(c[35*4],c[35*4+1],c[35*4+2],c[35*4+3],c[36*4],c[36*4+1],c[36*4+2],c[36*4+3],c[37*4],c[37*4+1],c[37*4+2],c[37*4+3],35);
                on stdcore[37]      : MergeFunction(c[36*4],c[36*4+1],c[36*4+2],c[36*4+3],null,null,null,null,null,null,null,null,36);
                on stdcore[38]      : MergeFunction(c[37*4],c[37*4+1],c[37*4+2],c[37*4+3],null,null,null,null,null,null,null,null,37);
                on stdcore[39]      : MergeFunction(c[38*4],c[38*4+1],c[38*4+2],c[38*4+3],c[39*4],c[39*4+1],c[39*4+2],c[39*4+3],c[40*4],c[40*4+1],c[40*4+2],c[40*4+3],38);
                on stdcore[40]      : MergeFunction(c[39*4],c[39*4+1],c[39*4+2],c[39*4+3],null,null,null,null,null,null,null,null,39);
                on stdcore[41]      : MergeFunction(c[40*4],c[40*4+1],c[40*4+2],c[40*4+3],null,null,null,null,null,null,null,null,40);
                on stdcore[42]      : MergeFunction(c[41*4],c[41*4+1],c[41*4+2],c[41*4+3],c[42*4],c[42*4+1],c[42*4+2],c[42*4+3],c[45*4],c[45*4+1],c[45*4+2],c[45*4+3],41);
                on stdcore[43]      : MergeFunction(c[42*4],c[42*4+1],c[42*4+2],c[42*4+3],c[43*4],c[43*4+1],c[43*4+2],c[43*4+3],c[44*4],c[44*4+1],c[44*4+2],c[44*4+3],42);
                on stdcore[44]      : MergeFunction(c[43*4],c[43*4+1],c[43*4+2],c[43*4+3],null,null,null,null,null,null,null,null,43);
                on stdcore[45]      : MergeFunction(c[44*4],c[44*4+1],c[44*4+2],c[44*4+3],null,null,null,null,null,null,null,null,44);
                on stdcore[46]      : MergeFunction(c[45*4],c[45*4+1],c[45*4+2],c[45*4+3],c[46*4],c[46*4+1],c[46*4+2],c[46*4+3],c[47*4],c[47*4+1],c[47*4+2],c[47*4+3],45);
                on stdcore[47]      : MergeFunction(c[46*4],c[46*4+1],c[46*4+2],c[46*4+3],null,null,null,null,null,null,null,null,46);
                on stdcore[48]      : MergeFunction(c[47*4],c[47*4+1],c[47*4+2],c[47*4+3],null,null,null,null,null,null,null,null,47);
                on stdcore[49]      : MergeFunction(c[48*4],c[48*4+1],c[48*4+2],c[48*4+3],c[49*4],c[49*4+1],c[49*4+2],c[49*4+3],c[56*4],c[56*4+1],c[56*4+2],c[56*4+3],48);
                on stdcore[50]      : MergeFunction(c[49*4],c[49*4+1],c[49*4+2],c[49*4+3],c[50*4],c[50*4+1],c[50*4+2],c[50*4+3],c[53*4],c[53*4+1],c[53*4+2],c[53*4+3],49);
                on stdcore[51]      : MergeFunction(c[50*4],c[50*4+1],c[50*4+2],c[50*4+3],c[51*4],c[51*4+1],c[51*4+2],c[51*4+3],c[52*4],c[52*4+1],c[52*4+2],c[52*4+3],50);
                on stdcore[52]      : MergeFunction(c[51*4],c[51*4+1],c[51*4+2],c[51*4+3],null,null,null,null,null,null,null,null,51);
                on stdcore[53]      : MergeFunction(c[52*4],c[52*4+1],c[52*4+2],c[52*4+3],null,null,null,null,null,null,null,null,52);
                on stdcore[54]      : MergeFunction(c[53*4],c[53*4+1],c[53*4+2],c[53*4+3],c[54*4],c[54*4+1],c[54*4+2],c[54*4+3],c[55*4],c[55*4+1],c[55*4+2],c[55*4+3],53);
                on stdcore[55]      : MergeFunction(c[54*4],c[54*4+1],c[54*4+2],c[54*4+3],null,null,null,null,null,null,null,null,54);
                on stdcore[56]      : MergeFunction(c[55*4],c[55*4+1],c[55*4+2],c[55*4+3],null,null,null,null,null,null,null,null,55);
                on stdcore[57]      : MergeFunction(c[56*4],c[56*4+1],c[56*4+2],c[56*4+3],c[57*4],c[57*4+1],c[57*4+2],c[57*4+3],c[60*4],c[60*4+1],c[60*4+2],c[60*4+3],56);
                on stdcore[58]      : MergeFunction(c[57*4],c[57*4+1],c[57*4+2],c[57*4+3],c[58*4],c[58*4+1],c[58*4+2],c[58*4+3],c[59*4],c[59*4+1],c[59*4+2],c[59*4+3],57);
                on stdcore[59]      : MergeFunction(c[58*4],c[58*4+1],c[58*4+2],c[58*4+3],null,null,null,null,null,null,null,null,58);
                on stdcore[60]      : MergeFunction(c[59*4],c[59*4+1],c[59*4+2],c[59*4+3],null,null,null,null,null,null,null,null,59);
                on stdcore[61]      : MergeFunction(c[60*4],c[60*4+1],c[60*4+2],c[60*4+3],c[61*4],c[61*4+1],c[61*4+2],c[61*4+3],c[62*4],c[62*4+1],c[62*4+2],c[62*4+3],60);
                on stdcore[62]      : MergeFunction(c[61*4],c[61*4+1],c[61*4+2],c[61*4+3],null,null,null,null,null,null,null,null,61);
                on stdcore[63]      : MergeFunction(c[62*4],c[62*4+1],c[62*4+2],c[62*4+3],null,null,null,null,null,null,null,null,62);
        }
        return 0;
}
```

8.6 Jump Table Used in the Ring Connection Architecture

```
while(!all_done)
select
{
    case cl :> input :
        switch(input)
        {
        case Spread_Data :
            SpreadData(pack,cl,cr,ct,Coreid);
            break;
        case Pass_On :
            PassingState(cl,cr,ps,g,offset,pack,0,DNEED,Coreid);
            break;
        case Pass_Free :
            printf("ERROR_Pass_Free core %d.\n",Coreid);
            break;
        case Core_Done :
            cl :> g[idD];
            cl :> done_num;
            if(g[idD]==Coreid)
            {
                if(done_num==Core_Num)
                {
                    cr <: All_Done;
                    cr <: 0;
                }
            }
            else
            {
                if(done==1)
                {
                    cr <: Core_Done;
                    cr <: g[idD];
                    cr <: done_num + done;
                }
            }
            break;
        case All_Done :
            cl :> all_count;
            if(Coreid==0)
            {
                if(all_count==0)
                {
                    cr <: All_Done;
                    cr <: count_total;
                }
                else
                {
                    go <: Stop;
                    go :> tb;
                     printf("NQueens            Core Use   Threads Board Size     Results Time_10ns      \n");
                     printf("All done   %d     %d   %d     %d     %d     %d     .\n",Core_Num,4,SIZE,all_count,tb);
                     printf("here!.\n");
                    all_done = 1;
            }
        }
        else{
            if(all_count==0)
            {
                cr <: All_Done;
                cr <: 0;
            }
            else
            {
                cr <:All_Done;
                cr <: count_total + all_count;
            }
        }
        break;}
    break;
case (int i=0;i<Thread_Num;i++)ct[i] :> thrid :
    ct[thrid] :> thrjob;
    switch(thrjob)
    {
    case Thread_Done :
        ct[thrid] :> count;
        count_total += count;
        cr <: Pass_On;
        cr <: Coreid;
        cr <: pack[0][currentD];
        cr <: 0;
        ps ++;
        PassingState(cl,cr,ps,g,offset,pack,1,NEED,Coreid);
        newp = g[pD] + offset;
        if(newp>pack[0][0])
            {
            inrun --;
            ct[thrid] <: Stop;
            if(inrun==0)
                {
                    done = 1;
                    cr <: Core_Done;
                    cr <: Coreid;
                    cr <: done;
                }
            }
        else
        {
            ct[thrid] <: GoOn;
            ct[thrid] <: pack[newp][leftD];
            ct[thrid] <: pack[newp][downD];
            ct[thrid] <: pack[newp][rightD];
            pack[0][currentD] = newp + 1;
        }
        break;
    }
    break;
}
```

8.7 Codes Used for Partitioning the Nqueens Problem

```
 par
 {
   GetDataPack(rec);
   GetDataReciever(pack,rec);
 }
 void GetDataPack(chanend rec)
 {
   GetDataSender(0,0,0,0,rec,0);
   rec <: Stop;
 }
 #pragma stackfunction 300
 void GetDataSender(int y, int left, int down, int right, chanend rec, int bit2)
    {
        int  bitmap, bit;
        if (y == Init_Depth)
    {
      rec <: GoOn;
      rec <: bit2;
      rec <: left;
      rec <: down;
      rec <: right;
     }
        else
     {
         bitmap = MASK & ~(left | down | right);
         while (bitmap)
         {
             bit = -bitmap & bitmap;
             bitmap ^= bit;
             GetDataSender(y+1, (left | bit)<<1, down | bit, (right | bit)>>1,rec,bit);
}    }    }
void GetDataReciever(int pack[][4], chanend rec)
{
   int num = 1,check = GoOn;
   while(check == GoOn)
   {
          rec :> check;
          if(check == GoOn)
          {
                 rec :> pack[num][bitD];
                 rec :> pack[num][leftD];
                 rec :> pack[num][downD];
                 rec :> pack[num][rightD];
                 num++;
          }
   }
   num--;
   pack[0][0] = num;
   pack[0][currentD] = 1;
   pack[0][coreD] = 0;
}
```

8.8 The Code used for Merge 4 Arrays Together

```
void Merge4Arry(short data2[], short data20[], short data21[], short data22[], short data23[], short s2)
{
       short n0,n1,n2,n3,i;
       for(n0=n1=n2=n3=i=0;i<s2;i++)
       {
       if(data20[n0]<=data21[n1]&&data20[n0]<=data22[n2]&&data20[n0]<=data23[n3])
              {
                     data2[i] = data20[n0];
                     n0++;
              }
              else
              {

 if(data21[n1]<=data20[n0]&&data21[n1]<=data22[n2]&&data21[n1]<=data23[n3])
                     {
                            data2[i] = data21[n1];
                            n1++;
                     }
                     else
                     {

 if(data22[n2]<=data20[n0]&&data22[n2]<=data21[n1]&&data22[n2]<=data23[n3])
                            {
                                   data2[i] = data22[n2];
                                   n2++;
                            }
                            else
                            {

 if(data23[n3]<=data20[n0]&&data23[n3]<=data21[n1]&&data23[n3]<=data22[n2])
                                   {
                                          data2[i] = data23[n3];
                                          n3++;
                                   }
                                   else
                                   {
                                          printf("ERROR_1\n");
                                          exit(1);
                                   }
                            }
                     }


              }
       }
       data2[i] = Max_Num;
}
```