

Abstract

The key word of my project is parallel programming. As the CPU core is becoming faster and faster, it is still unable to meet the need of real applications. The idea of combining some low performance processors together to build a super computer had been proposed for decades. Recently more and more high- performance parallel computer systems have been developed. In some fields multiprocessor systems are commonly used, and multi-processors will be a trend in the next few years.

Parallel programming works on the principles that large problems can be divided into small ones, by spreading the small tasks to all the processors. By combining all the processed data, the multi-processors system can solve problems quicker and more efficiently. Many programming algorithms exist, each of them with its own strength, some require the support of specific organisation of hardware.

The ARM processor is a 32-bit reduced instruction set computer (RISC) instruction set architecture (ISA) developed by ARM Holdings. It was known as the Advanced RISC Machine, and the ARM processor is one of the most popular microprocessors in the world. Therefore combining the parallel programming technique with ARM processors is tempting.

To increase the efficiency of parallel algorithms on ARM processors, evaluation of current algorithms is needed, and some modification to the algorithms will be made.

In this project, I first researched several selected algorithms about their possibility to be parallelized, and what is the potential improvement.

By paralyzing FFT algorithm, experiments results show the advantages of message passing approach compared with shared memory approach.

Although this project reached certain extend of success, there is still improvement space. Suggestions for the future development of this project are also given lastly.

Acknowledgements

I would like to express my deep gratitude to Dr Simon Hollis for his help, interesting discussions and constant encouragement through all the stages of this project.

I would like to express my deep appreciation to my family. Whilst I have not seen them for a whole year whilst doing this course, their encouragement and support always were always with me.

I also thank them for the financial support without which I could have never taken this course.

Abstract	I
Acknowledgements	II
Declaration	III
Chapter 1: Introduction and Context	- 1 -
1.1 Aims and Objectives	- 1 -
Chapter 2: Introduce parallelism and background	- 3 -
2.1 Amdahl's law	- 3 -
2.2 Taxonomy	- 5 -
2.3 Distributed-shared-memory Architectures	- 7 -
2.4 Memory coherence	- 9 -
2.5 Parallel Programming Model-Message Passing	- 11 -
2.6 Parallel Programming Model-Shared memory	- 14 -
2.6.1 Basic Directory-Based Cache Coherence Protocol	- 14 -
2.6.2 Cache coherency	- 15 -
Chapter 3: Project implementation	- 19 -
3.1 Previous work and new tasks	- 19 -
3.2 Methodology	- 20 -
3.2.1 Principles of Parallel Algorithm Design	- 20 -
3.2.2 How the execution time is calculated.	- 21 -
3.2.3 How to map the task to processors	- 23 -
3.2.4 Case study	- 24 -
3.3 Software and target multi-core device	- 26 -
3.4 Algorithms selection	- 27 -
3.5 Sequential sum	- 29 -
3.6 Bubble sort	- 31 -
3.7 FFT	- 32 -
3.7.1 Principles of FFT	- 32 -
3.7.2 Implement FFT in serial	- 34 -
3.7.3 Theoretical timing analysis	- 35 -
3.7.4 Parallelize FFT using message passing approach	- 36 -
3.7.4 Parallelize FFT using shared memory approach	- 41 -
Chapter 4: Conclusion	- 45 -
Chapter 5: Future Development	- 48 -
5.1 ARM Corte-9x Mpcore	- 48 -
5.2 Parallelize FFT	- 49 -
Bibliography	- 51 -
Appendix: Source code:	- 53 -

Chapter 1: Introduction and Context

Traditionally, computer software is written in serial, all the instructions are constructed and implemented as serial sets of instructions. All the instructions may wait in a queue first, then the processor will execute one instruction at a time, after the current instruction is finished, the next one is executed.

Parallelism is first being introduced into the architecture of computers [20], pipelining, functional, array and multiprocessing. The first three architectures are specifically designed, which is outside the scope of this dissertation.

Multiprocessing means several processors each obeying its own instructions, and usually communication via a common memory, or connected in a network. Parallel computing uses multiple processing elements to solve a problem concurrently. One big problem is broke into separate parts, and each processing elements can process its part of the algorithm.

Parallel algorithm is a kind of algorithm which can be executed one part at a time on different process elements, and feed back the result to get a right answer. For example, the task is using the prime-sort algorithm to sort numbers and find the prime numbers amount them. In parallel programming, we can divide the 100 into ten parts, each process elements can operate on one part of the task. This kind of algorithms can be easily divided, but some other kind can't. Sorting, searching and signal processing are the common parallelizable tasks.

1.1 Aims and Objectives

Not like the other special architectures, ARM processors are not designed to perform parallel computations, The ARM 7 has one of the simplest possible pipelines. It is three stages long, and very straightforward, ARM9 pipeline is more complex, containing 5 stages. In pipelined architectures, data hazards occur if you attempt to read from a register that is about to be updated, but has not quite been.

The aim of the project is to research, design and derive parallel algorithms run efficiently on multi-ARM processors. This project uses Realview developing suit v4.1 to simulate the performance of multi-ARM processors.

The objectives to meet this aim can be broken down as follows:

1. Study some different exiting algorithms, how they divide one large task into small tasks, and mange these tasks, how to map them to multi-processors, how to allow execution of maximum parallel tasks.
2. Analysis the data flow, study the possibility to parallelize the serial algorithm.
3. Implement the selected algorithms in serial and observe the performance.
4. Implement the selected algorithms using message passing and shared

memory methods.

5. Evaluate the performance of different parallel algorithms on multi-ARM processors.
6. Try to modify the algorithms to enhance the performance.
7. Analysis the results and propose improvements.

Chapter 2: Introduce parallelism and background

2.1 Amdahl's law

Amdahl, Gene in 1967 proposed a method of finding the maximum theoretical speedup using multiple processors.[9] According to Amdahl's law, also known as Amdahl's argument, "The maximum speedup (the maximum number of processors which can be used effectively) is the inverse of the fraction of time the task must proceed on a single thread." [4]

If one task has to be implemented serially 2% of all the executing time, the maximum number of processors is 50, the result will be 50 times faster. In Amdahl's law is

$$Max_speedup = \frac{1}{\frac{Fraction_{improved}}{Speedup_{improved}} + (1 - Fraction_{improved})}$$

It reminds that "serial dependency" is one of the big hurdles. There always limited parallelism available in the task. Like shown in the equation, in the reality, the fraction can be improved is quite small.

"To achieve a speedup of 80 with 100 processors, only 0.25% of original computation can be sequential." [1] One of the other challenges is the high cost of communications between processes (which will be detailed discussed later). Almost all parallel applications require their processes to communicate, the communication includes sharing intermediate results, synchronizing state, merging final results, etc.

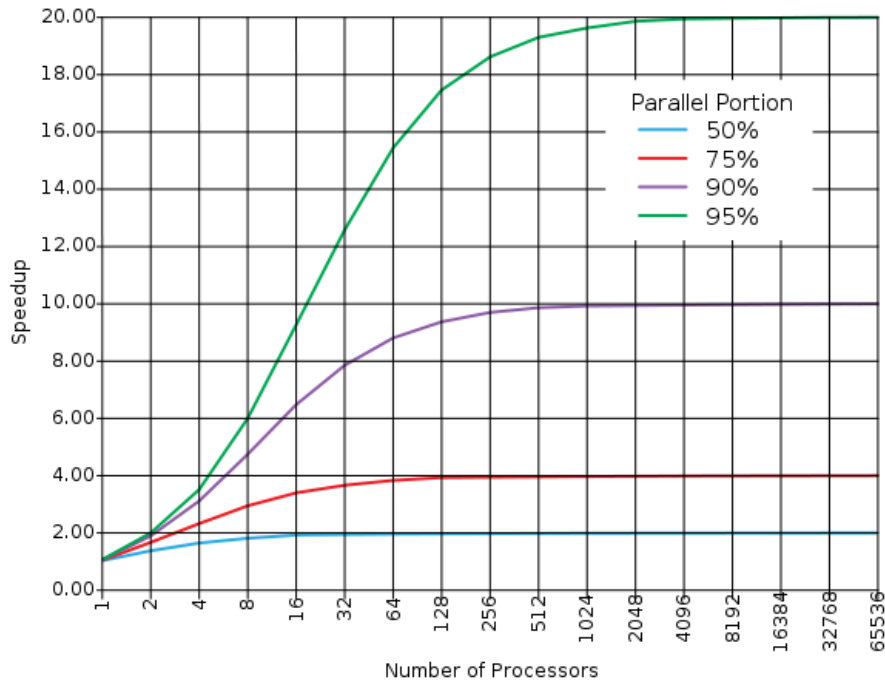
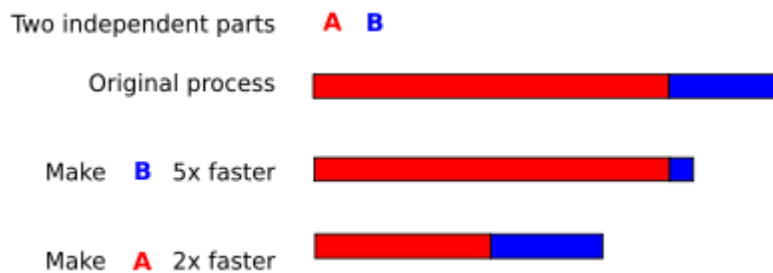


Figure1. Amdah's law

As we can see from the figure, the speed-up of a program from parallelization is limited by how much of the program can be parallelized. For example, if 90% of the program can be parallelized, the theoretical maximum speed-up using parallel computing would be 10x no matter how many processors are used. In a sequential program,

$$\text{The maximum speedup is } \leq \frac{p}{1 + f(p-1)} \quad (1)$$

(P is the proportion of a program that can be made parallel, f is the fraction of time spent in the part that was not improved)


 Figure2.
(Source: [9])

For example, as we can see in figure 2, in one program, there are two independent parts A and B, it takes about 25% time to compute part B, so when we make B 5 times faster, the whole speedup by equation 1 is

$$\text{Maximum speedup} \leq \frac{5}{1 + 0.75(5-1)} = 1.25$$

However if we make part A just 2 times faster, the whole speedup is:

$$\text{Maximum speedup} \leq \frac{2}{1 + 0.25(2-1)} = 1.6$$

It appears that, it is better we just make part A 2 times faster. So in the experiment later, we should take more attention on which is the most time consuming part of the sequential program.

2.2 Taxonomy

According to Flynn [1966], he placed all computers into one of four categories:[1]

1. Single instruction stream, single data stream (SISD).

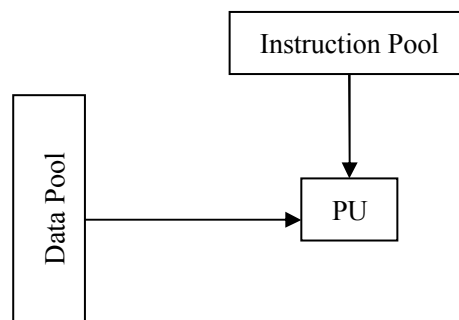


Figure3. SISD

2. Single instruction stream, multiple data stream (SIMD)

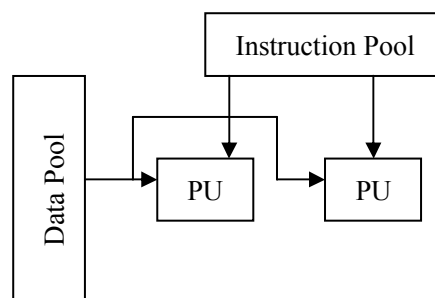


Figure4. SIMD

3. Multiple instruction streams, single data stream (MISD)

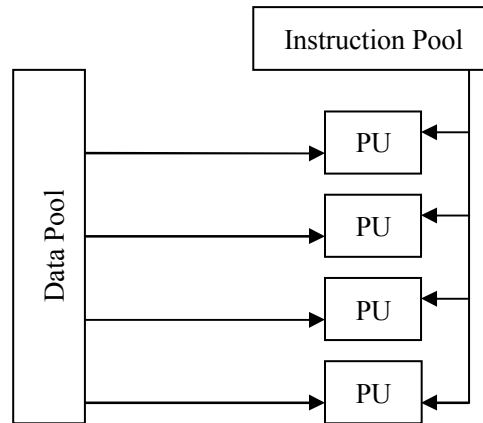


Figure5. MISD

- 4.
5. Multiple instruction streams, multiple data stream(MIMD)

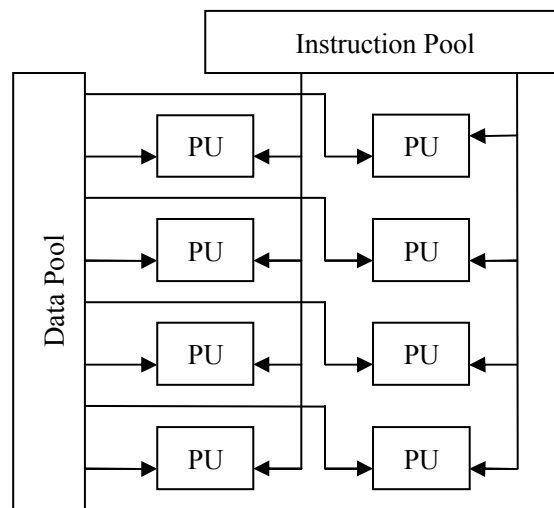


Figure6. MIMD

In this category, SISD is the traditional uniprocessor, single instruction stream, single data stream. In SIMD one instruction is executed by multiple processor elements, each of them has its own data flow and memory. There are different classes of processors in this category, DSPs is one of them. DSPs have different operating cores, add unit, multiple unit and read unit... but still DSPs use single memory and control processor fetch and dispatch instructions, and then implemented in different units. MISD, there is no commercial multiprocessor of this type. [1]

MIMD, computers use MIMD usually have a number of processors, each of them may execute different instructions and has its own data. Different processors can operate on different part of the program, which is the specialty of this architecture, it can work on shared-memory and distributed memories, in shared-memory, MIMD need communication between processes and between processors, it can be bus-shared or hierarchical type. "Most modern MIMD machines use an explicit message passing paradigm to communicate

between processors. it also gives the programmer the control needed to minimize communications and maximize performance.” Affan Ahamd said.[2] Here is where Flynn stopped.

Various extensions of taxonomy had been proposed by others. According to S. Tanenbaum, MIMD computers can be divided into two groups [3]: shared memories called multiprocessors and those have private memory called multi-computers.

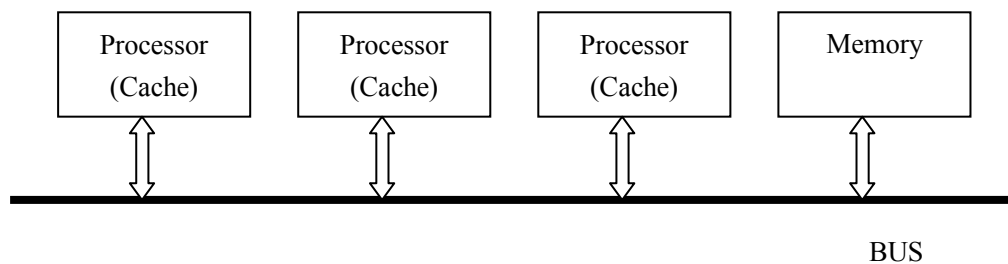


Figure 7. Bus-based shared-memory multiprocessors.

As showed in figure 5, is the bus-based shared-memory multiprocessors. All the processors are connected to a “global bus”, via this bus all the processors can access the global memory. The architecture is easy to understand, and easy to build, it is just a extension of the uniprocossor system, but it will face the problem of memory coherence (which will be discussed later), that add a lot of difficulties in the OS(Because the OS has to manage the memory coherence problem). In this kind of MIMD, because of the specialized architecture, the number of processors is limited, the maximum is a few dozen.[6] When the number of processor grows, the complexity of shared-memory management will increase significantly.

The other kind of MIMD is distributed-memory architecture. And because of its architecture-physically distributed memory, it can support larger number of processors. As the rapid growth in the number of processors in a system, the bus-based shared-memory system will meet a bottleneck [7], the process will wait for a long time to access the memory. So distributed-memory is more efficient in the situation the processors require more memory bandwidth and lower memory latency. This part will be detailed discussed in the next chapter.

2.3 Distributed-shared-memory Architectures

Distributed-memory is easy to build, because unlike shared-memory machines they do not require complex and expensive hardware cache controllers [Archibald and Baer 1986]. The Distributed shared memory(DSM) is first proposed by Li in 1986 and Li and Hudak in 1989.[4] There is a large piece of

global memory which can be accessed by all the processors, and at the same time, each processor gets its own private non-shared memory. It is a distributed memory model that works on a shared-memory machine. DSM combines the advantages of shared-memory and distributed-memory architecture. It supports the easy program on shared-memory and is less expensive than the distributed architecture.

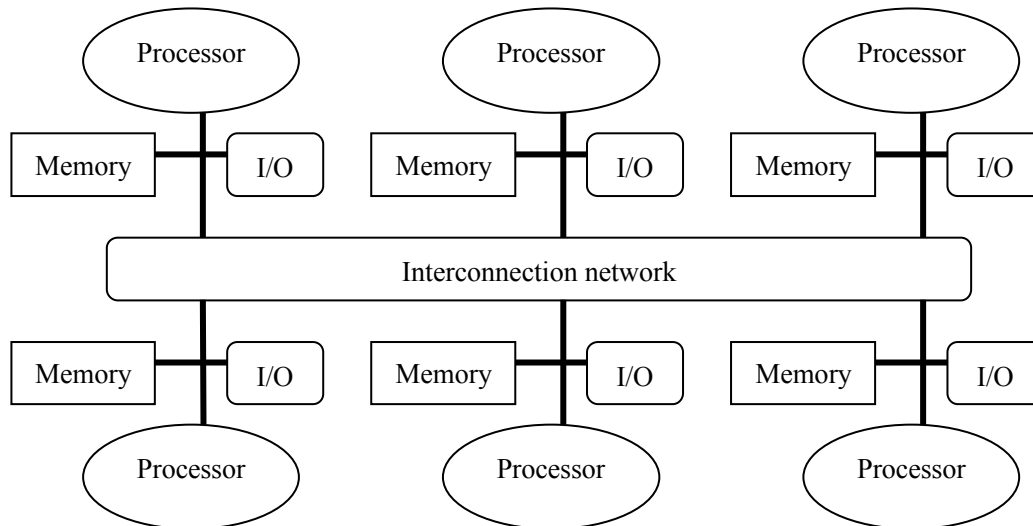


Figure 6. The basic architecture of distributed-memory multiprocessors
(Source: [1] Page 532)

As shown in Figure 6, memory is distributed among the nodes and all nodes are connected by an interconnection network. The basic idea is that, each processor has its own memory, and can only do operations on its local data, but when one processor needs a remote data (which the data is not stored in its memory), the processor will communicate through the interconnection network with other processors, and because of that, it is more effective to scale the memory bandwidth, and reduces the latency for accesses to the local memory.[1]

However, the disadvantage of a distributed-memory architecture is that the communications between the processors. It is more complicated and time consuming.

The biggest problem on DSM is the large amount of communications it needed to maintain the consistency of the memory data. [7] The inflexible consistency protocol and restrictive memory model make it difficult to reduce the communication. In the recent research, many approaches have been proposed to reduce the communication. "Software release consistency" [Gharachorloo et al. 1990], "Multiple consistency protocols" [Bennett et al. 1990; Eggers and Katz 1988]...

2.4 Memory coherence

When two or more processors are trying to access and change the value at one same memory location, there will be problems about the memory coherence. If one process cached the data of one piece of memory, and another process updated the value stored in the memory, the processes which have cached the same memory will be using the invalid value without notification.

For example, consider the following program:

Processor A	Processor B
A1: Read the value X	B1: Read the value X
A2: ADD 1 to data X	B2 : ADD 1 to data X
A3: Store X back to memory.	B3: Store X back to memory.

If the instruction B1 is executed between A1 and A3, or instruction A1 is executed between 1B and 3B, the program will produce error data. To solve this problem, we can simply just add one lock to the value X, when processor A is accessing the data, then the data is locked, that means processor B can't access to the data X. This method can fix the problem, but at the same time, it will slow down the computation.

In the parallel computing, there is another problem, like this bank withdraw algorithm:

```
1      int withdraw ( int withdraw )
2      {
3          if (balance >= withdraw)
4              {
5                  balance=balance- withdrw;
6                  return 1;
7              }
8          return 0;
9      }
```

If the machines find that $\text{balance} \geq \text{withdraw}$, they will let the customers withdraw the money. However, suppose the balance is 800, and there are two people try to access the bank account at same time, and try to withdraw 500 and 600 respectively. The machines will find out that all $\text{balance} \geq \text{withdraw}$, and return 1. Since both processes performed their withdrawals, the total added up withdraw is more than the balance.

So there will be need a “lock” and “unlock” instructions in this algorithm

```

1    int withdraw ( int withdraw )
2    {
3        When one process accesses the account, lock;
4        if (balance >= withdraw)
5        {
6            balance=balance- withdrw;
7            return 1;
8        }
9        return 0;
10       Finish action, unlock;
11    }

```

The “lock” and “unlock” mechanism insured that only one process is manipulate the data. Although, it avoids memory coherence, it slows down the computation at the same time.

In Finite Difference problem, there are several serial tasks, each of them is related to the right hand side and left hand side, only one value is stored in one task. Each task has two input and two output, no task will be updated to step $t+1$, until the two task next to it is updated to step t .

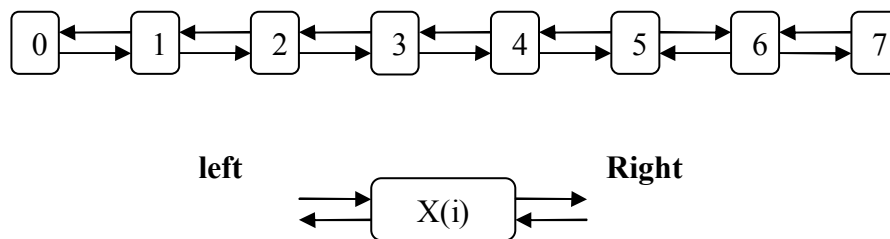


Figure18.Finit Differences example tasks
(Source:[14] Section 1.4.1)

As shown in figure 13, $N=7$, every node is connected to two nodes, it has to communicate with these two, before update itself. The equation is :

$$0 < i < N-1, 0 \leq t \leq T; X_i^{t+1} = \frac{X_{i-1}^t + 2X_i^t + X_{i+1}^t}{4}$$

As this task is going, the value stored in each node should repeatedly update. Each processor is responsible for one node. There are three jobs for each processor. 1. Receive the value from left and right processors (nodes). 2. Calculate the updated value $X(t+1)$. 3. Send the value to the left and the right nodes.

By this method, N processors can execute on its own, and because of the value should be calculated according to the value next to the node, the task will be synchronized by the receiving function.

2.5 Parallel Programming Model-Message Passing

Parallel programming introduces additional source of complexity.[14] Not only the number of instructions will increase, but also the programmer should manage the communications and cooperation between the processors. So abstraction and modularity should be the key to the parallel programming.[14]

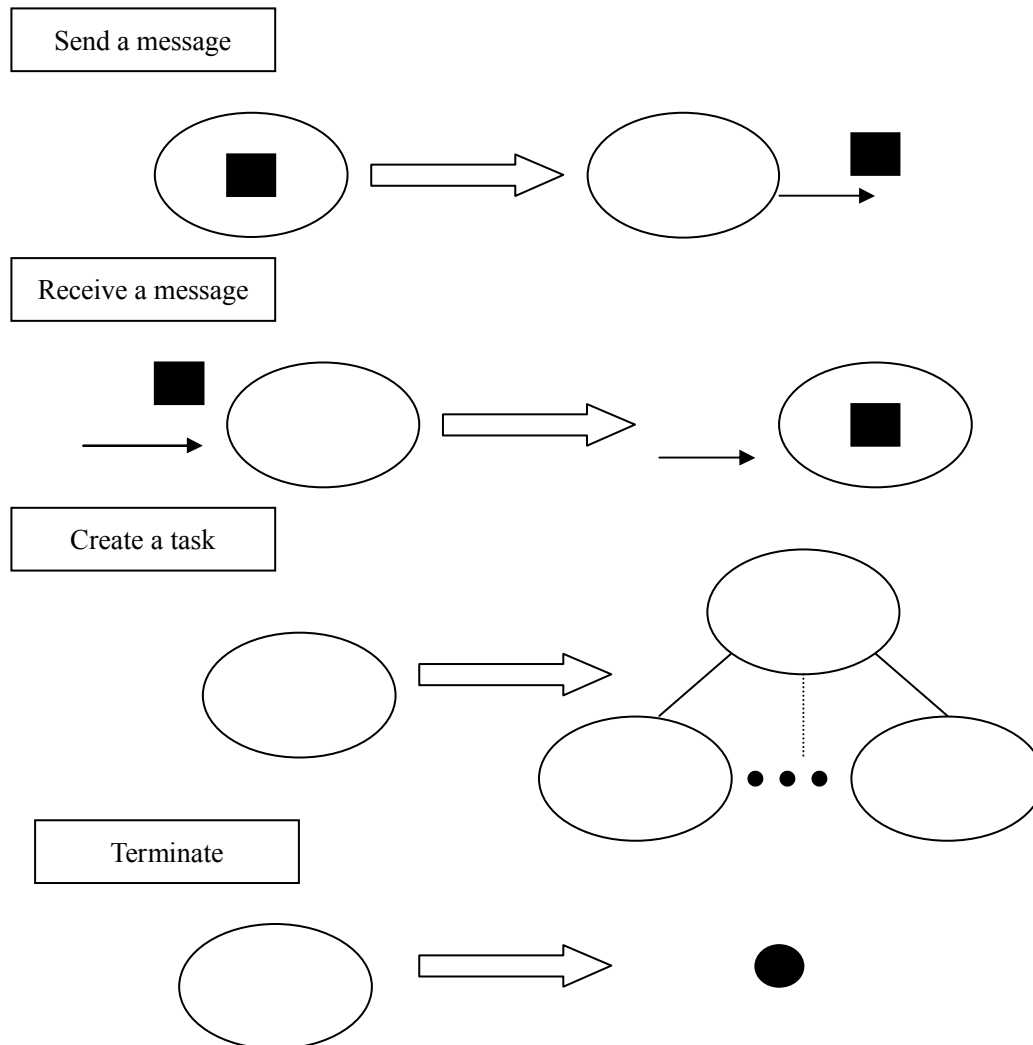


Figure10. Four basic task actions
(Source:[14] Section 1.3.1)

One task in the parallel programming can do these four basic task plus read and write. Channels between processes are very dynamic [14], it can be created as the message is sending (means the channels can be created in the code of sending messages).

Message passing has been widely used in the area of parallelism and inter-process communication. The key of the parallelism is communication. In order to organize all the processes, the multiprocessors-system must have

the ability to communicate and synchronize. One way is to share the memory, several processes can access to the same memory, which may cause the memory coherence problem, and the other way is message passing. By message passing, one process can send information to another. There are several ways to manage message passing. One process can request the operating system to pass the message to another process, so the after receiving the message, the process can use the data directly. In some other system, the process will deposit and retrieve the messages from named “pickup points” [8].

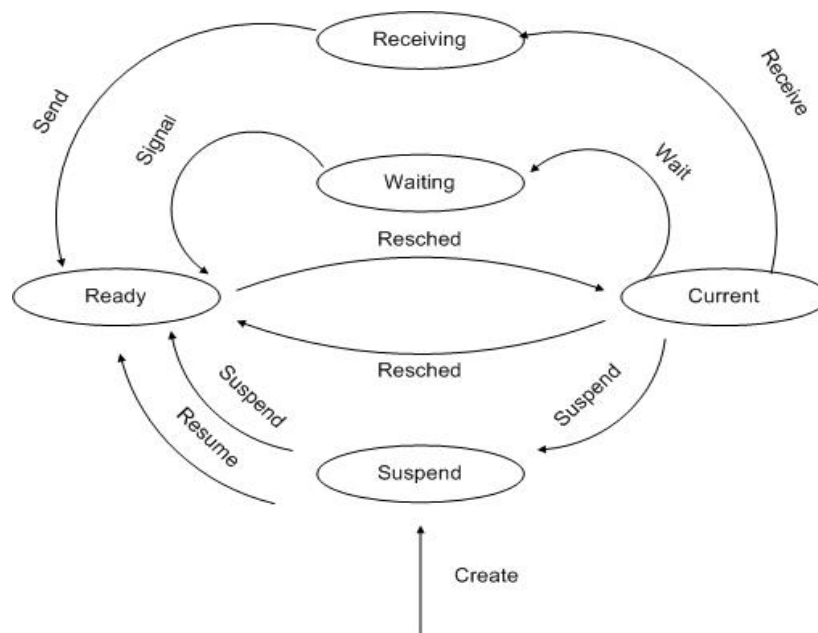


Figure11. Process state of the receiving state transitions.

(Source: [8] Page 95)

In the message passing mechanism, message passing functions link the library to make function calls to receive and send message. One process will send explicit messages to one address or several addresses, and the destination will save and buffer the message, and notice the sender that the message has been received.

In Figure 11, is the receive state of the process transition in Xinu.[8] There are three manipulate messages :*receive*, *recvclr*, and *send*. The arguments of the “*send*” are message and process id. *Receive* will wait for a message to come and then send a return message to the sender, there is no argument. *Recvclr* is a message used to clear away old waiting messages.

There is another problem discussed in [8] page 97. Where should the sender store the messages? First the message can’t be stored in the sender’s memory, because the sender which is a process may finish before the message is arrived and received, if the process is finished, the memory may be cleared by other process. Second the message can’t be stored in the receiver’s memory, because that means one process can change the data of

another process's memory, that is a very dangerous behavior. So the message should be stored in a "pickup points", the senders will check if the receiver have a message waiting, if not , the sender will deposit the message, and notify the receiver that there is a message waiting, finally the receiver can get the message successfully.

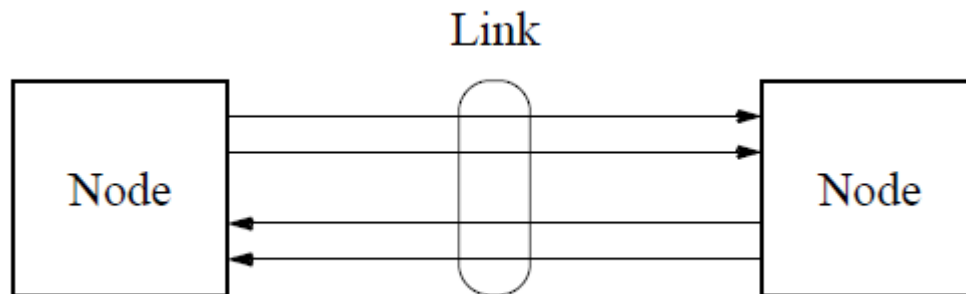


Figure12. Links between two nodes

(Source: 14)

As we can see in figure12, in the message passing approach, the communication is between two nodes, the links have two directions, so that the nodes can communicate with each other. Software and hardware overhead in sending message and the actual transmission time is the total time to send message, including software overhead and interface delays.

By reading the code in [8] Page 96 and page 97, the sender will first check the data of `(pptr=&proctab[pid])->pstate==prfree` , that means the receiver has no message waiter for it to pick up. Then deposit the message by store the pointer of `pptr->pmsg`, then indicates `pptr->phasmsg++`, the specific receiver has a message waiting, when there is another message coming, it should wait first. The receiver can receive the message by receiving the message pointer: `msg=pptr->pmsg`.

2.6 Parallel Programming Model-Shared memory

2.6.1 Basic Directory-Based Cache Coherence Protocol

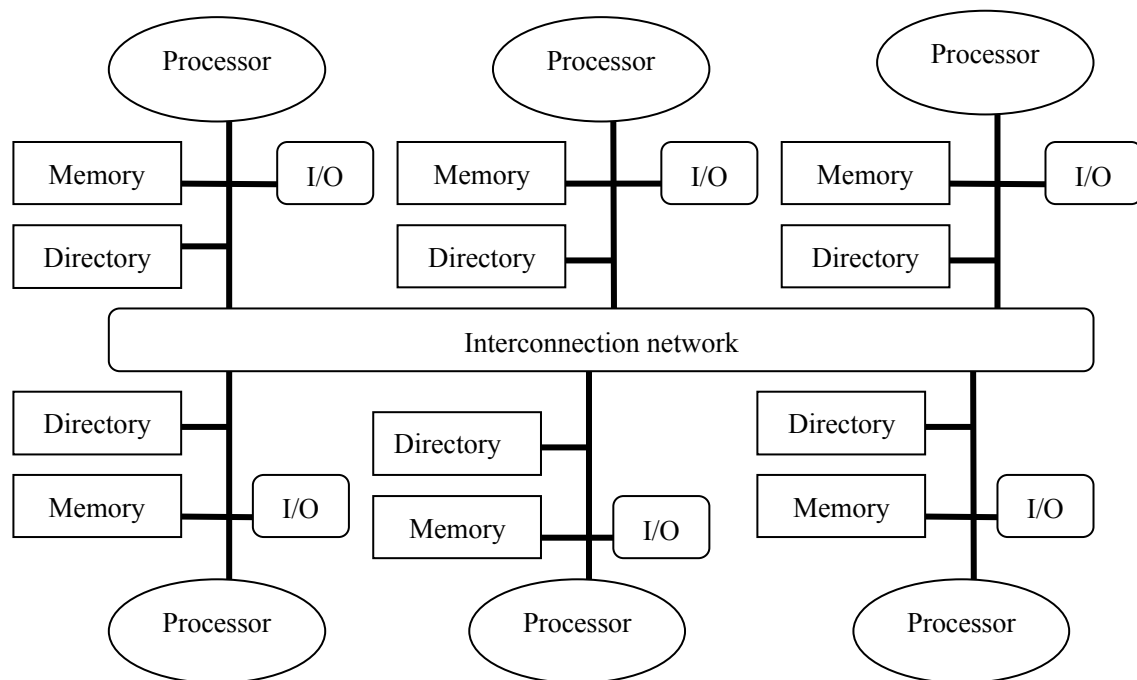


Figure 8. Directory-Based Cache Coherence Protocol

(Source: [1] page 578)

Comparing Figure 8 and Figure 6, each node has one directory. Each directory is tracking the caches that shared same memory, it contains a number of pointers to the block of shared memory. [13] The directory can communicate with the processor. The processor should first ask the directory for permission to load data from the private memory to its cache. When the data is updated, the directory is first notified, then it can update the data in the cache or invalidated it. [19]

There are 7 possible messages sent among nodes by directories:

“Read miss” means one processor has a read miss at a place, it requests data to be shared. “Write miss” means one processor has a write miss at a place, it requests data, and updates the data in the location.

“Invalidate” invalidates a shared copy of data.

“Fetch” invalidates to fetch the address and sent it back to the original directory and changes the state to shared.

“Fetch/invalidate” invalidates to fetch the address and sent it back to the original directory and change the state to invalid.

“Data value reply” invalidates return a data value from the original address.

“Data write back”, write back a data value. [13]

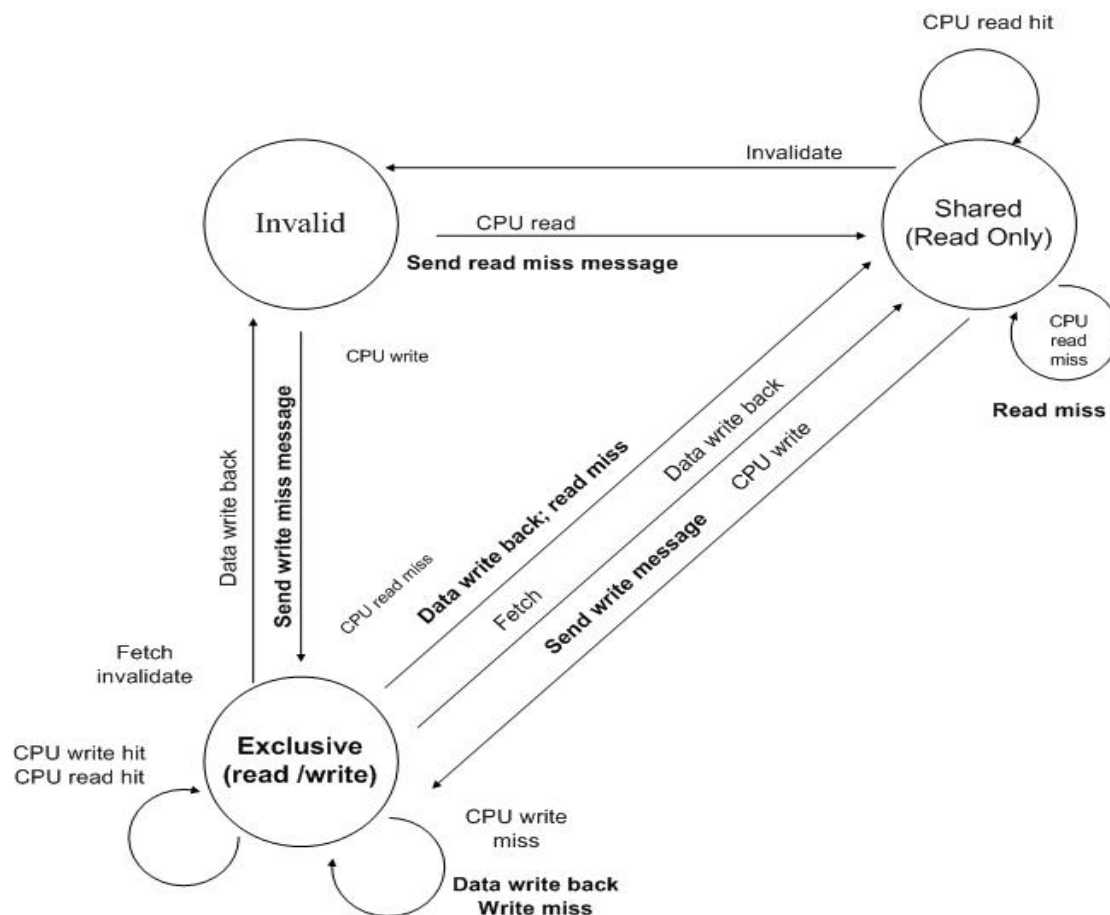


Figure 9.State transition diagram for an individual cache block
(Source: [1] page 581)

Compare this state transition diagram with Snooping protocol, the main state of the directory is the same as in Snooping protocol. The write miss message was broadcast to every processors by bus in Snooping protocol, in directory-based memory, the directory stored pointers, which point to all the caches shared the same memory, so the message is sent by the directory selectively. [19]

Only the nodes which have cached the same memory will get this message. When a directory gets a message, the directory will not only update the state of it but also sent an additional message to operate the message. And the state of a directory represents not only one node, but also represents all the cached copies of a memory address.

2.6.2 Cache coherency

It is a new way of exchanging data between programs running at the same time. In shared memory approach, one process will create an area in RAM which other processes can access, so the several processes can share the data.

Since all the process can access the shared memory like their own distributed memory (mentioned in section 2.3), it is a very fast way to communicate with each other. But it will also cause another problem called cache coherency.

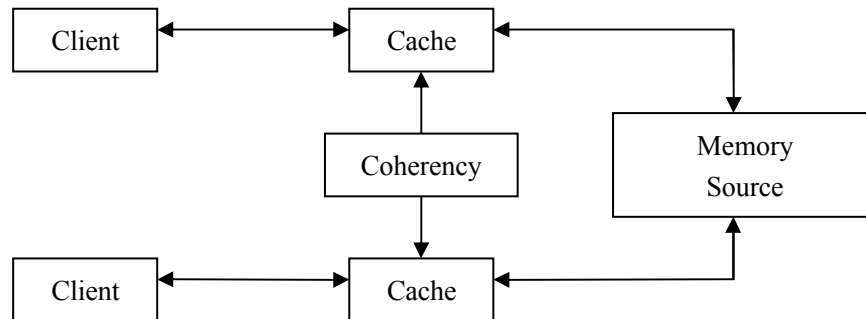


Figure 7 Cache Coherency

In Figure 7, all the processors have caches, because of the coherence problem, the shared part memory cannot be cached, and only the private memory can be cached. According to Tartalja(1996) the software can cache the value of shared data by copying the data from the shared address to a local address, and then be cached.[6] This approach was discussed in [1] page 567, it has three main disadvantages and large latency of access to remote memory.

So “cache coherence is an accepted requirement in small-scale multiprocessors”. [1] Because it needs the coherence protocol to coordinate the cache and the memory, the data provide by the cache and memory should be consist. As defined in the system’s memory consistency model.[10]

Each cache line is in one of these states: dirty (the data has been updated), valid (the cache line is current), invalid and shared. Each individual cache will watch the bus, when a quest of writing the memory location where the cache have already loaded, the cache will use the bus to broadcast the request to all processors, the cache will indicate the data to be invalid(dirty).

Compared with directory-based mechanism, Snooping is simpler and faster, because all the messages are broadcast through the bus, all the nodes can receive and send the messages.

For example, the caches have 3 extra bits

V: valid

D: Dirty bit, signifies that data in the cache is not the same as in memory

S: Shared

	ID	V	D	S
1111	00	1	0	0
0000	01	0	0	0
0000	10	1	0	1
0000	11	0	0	0

When a write occurs in address of 1111 00

	ID	V	D	S
1111	00	1	1	0
0000	01	0	0	0
0000	10	1	0	1
0000	11	0	0	0

The disadvantage of Snooping is that it needs large amount of bandwidth, as discussed in [1] page 577, 16 processors, block size of 64 bytes, 512 KB data cache, the total bus bandwidth demand is 1 GB/sec for Barnes and 42 GB/sec for FFT. And at the same time, the system which is using Snooping protocol is not scalable. Every request to access the memory will be broadcast to all the nodes in the system, so when the system gets larger, the bandwidth demands will get larger too, at the same time the bandwidth of a bus is limited.

There is another way to solve the cache coherency problem is by bypassing the cache.[23] When write data,

1. If (the cache is clean==0)
2. clean the cache and write unprocessed-data to memory
3. If (cache is clean==0)
4. Read the new data in cache
5. Read the old data from memory
6. Clean the cache and write new data to memory
7. If (cache is clean==1)
8. Store the old data in cache
9. Read the old data in cache line [23]

The steps described here can be illustrated

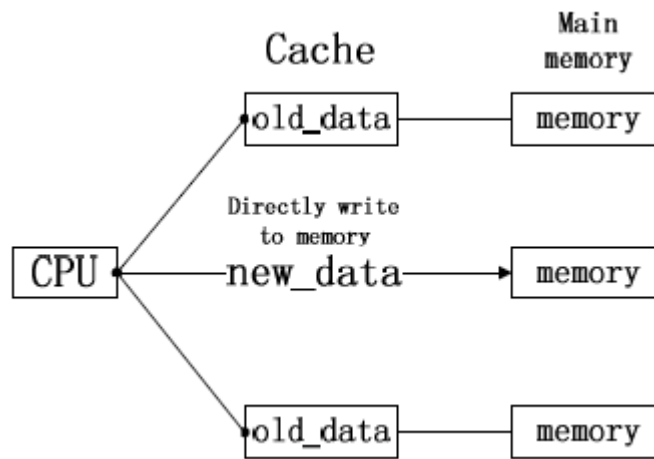


Figure8. Write data without cache.

Chapter 3: Project implementation

3.1 Previous work and new tasks

In terms of the hardware, the Distributed shared memory (DSM) is first proposed by Li in 1986 and Li and Hudak in 1989. DSM is the most common multi-processors architecture. It is the combination of distributed memory and share-memory, there is a large global memory that every processors can access, and every processors have its own private memories. DSM combines the advantages of shared-memory and distributed-memory architecture. It support the easy program on shared-memory and less expensive than the distributed architecture. Because DSM, this project is using, shared-memory architecture will inevitable face the problem of memory coherence.

The two most widely discussed coherency problems are snooping and directory-based, each have its own strength and weakness. Snooping is more simple and faster, because snooping is built based on the uniprocessor system, but snooping requires large mount of bus bandwidth. Directory-based system, it stores pointers in the directory, the directory will send the message selectively instead of broadcast the message to all caches.

In terms of the algorithms, because of the limitation of “serial dependency” which is hard to change, parallel algorithm usually used in the field of sorting, searching and signal processing. FFT is the most common algorithm used in signal processing, Cooley–Tukey proposed Cooley–Tukey algorithm in 1965, which is the first FFT algorithm, and there are a lot of improvement nowadays. FFT on shared-memory architecture is analyzed by A. Norton and A. J. Silberberger.(1987) Cui-xiang(2005) reduced the complexity of FFT to $O((n \log_2 n) / p)$, P is the number of processors.

The FFT implementation state of the art:

- Hypercube architecture [24, 13, 17]
- Arrays architecture [12]
- Mesh architectures[31]
- An instructive shared-memory FFT for Alliant FX/8 [29].
- Additional performance studies on shared-memory FFT [21]
- Memory hierarchy to an effective FFT implementation[14]

- CRAY-2 [6]
- EARTH, fine-grained dataflow architecture. [20]

In this project, I will first evaluate the algorithm of FFT on the multi-ARM processors, familiarize the idea of algorithm, how the algorithm divide the problem, and map the small tasks to the processors, then try to improve and design algorithm based on the ARM architecture.

3.2 Methodology

3.2.1 Principles of Parallel Algorithm Design

In order to obtain the optimum performance from any computer it is necessary to tailor the computer program to suit the architecture of the computer. [20]

Parallel algorithm is a kind of algorithm which can be executed one part at a time on different process elements, and feed back the result to get a right answer. For example, the task is using the prime-sort algorithm to sort numbers and find the prime numbers amount them.

In parallel programming, we can divide the 100 into ten parts, each process elements can operate on one part of the task. This kind of algorithms can be easily divided, but some other kind can't. Sorting, searching and signal processing are the common parallelizable tasks.

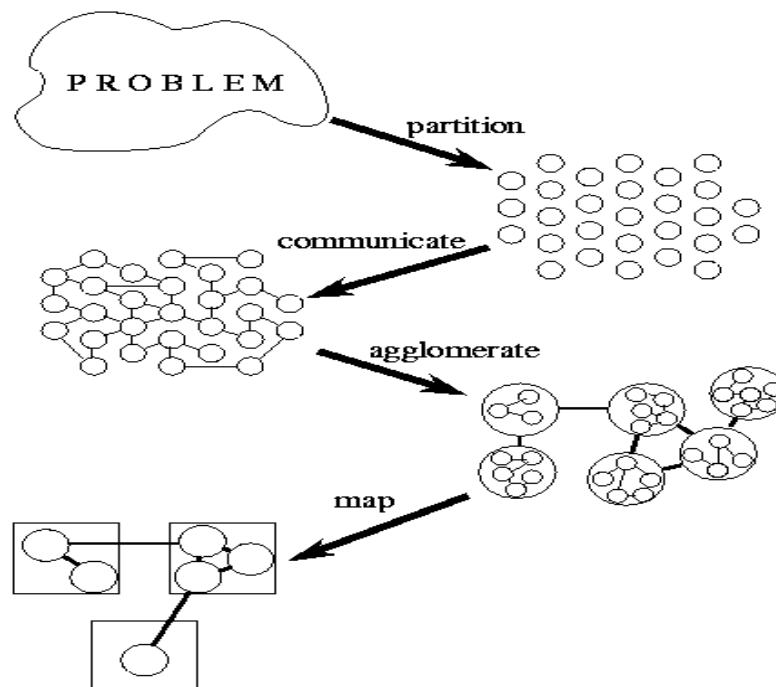


Figure12. A design methodology for parallel programs
(Source: [14] Section2.1)

Parallel programming will start with the problem specification, then separate it into parts, find out the resources that different part requires, synthesis them together, at last map them to separate processors.[14]

The first problem is how to divide a problem efficiently. The focus should be how we can find a large mount of small tasks to complete one big task. The method should provide flexibility in terms of the parallel algorithm,[14] and later reduce the communication between processors. Dividing tasks should not just concerning about the computation tasks, but also the data associate with it. After the task is divided, all the processors should have equal amount of work to do. Not only the number of instructions, but also the data which the processor is calculating, that means that memory access time should be equal. Another way to divide the task is by function. When we trying to divide a task, and when we trying to verify the method we are using, the data we are using may be disjoint. So in this case, we can jump over the data and instruction level, go straight forward to the function unit. Divide the tasks into different function unit, each individual processor can be mapped to in charge of one of its function.

3.2.2 How the execution time is calculated.

The idea of parallelism is to have multiple processors assigned to a single task that has been divided into several concurrent threads. In theory, this may shorten the total computation time by the number of processors used (linear speedup).

In practice, however, the performance of parallel programs is strongly dependent on the data flowing, the communication effort between the concurrent threads.

One of objectives of the project is to evaluate the performance of the exiting parallel algorithms' work. The performance of a parallel program is complicated. We should consider the execution time, costs in the software cycle, parallel scalability, memory requests, communication times between processors, memory access miss times....etc.

In this report, I focused the attention on how the execution time is calculated. There are three kinds of execution time I am going to test: idle time, communication time, and computation time.

Idle time is when the processor is waiting for the message to receive or send, or waiting for the remote to come, or waiting for the data which the processor needs to proceed.

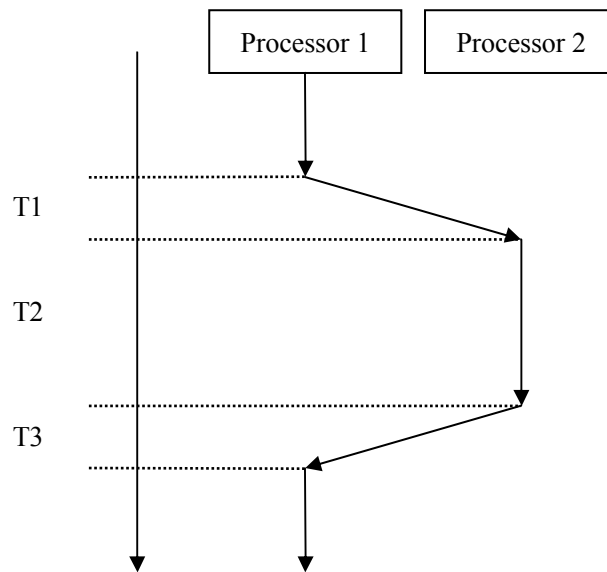


Figure14. Execution Time

In figure14, T1 and T3, processor 2 is waiting for the message passing from processor1 and sending the message. Meanwhile, during T2, processor 2 is computing, processor 1 is waiting for the data in processor2. The idle time for processor1 is $T(\text{idle})=T1+T2+T3$; for processor 2 $T(\text{idle})=T1+T3$. Before T2 time comes, processor2 can full the time by loading the instruction and data, or do next instruction. Another way to reduce the idle time is to create multi-task for one processor.[14] When one process is waiting for the messages or waiting for the data to come, the processor can automatically switch to another process.

Computation time is the time processor doing the computation rather than doing the communication and idle. [14] It is easy to record. Usually we are trying to parallel a serial algorithm, and the serial algorithm which performs the same task is the computation time.

Communication time, is the algorithm spending on sending and receiving messages.[14] There are two kinds of communication costs, inter-processor communication and intra-processor communication.[14] Inter-processor communication means the kind of message passing between two different processors, on the other hand, intra-processor communication is the kind of message passing between two processes, which means it is inside one processor. In most cases, the intra communication is much faster, because it happens in the local memory even in the cache, and inter communication happens between memory and memory, the message sometime has to pass the interconnection network.

There are two parameters:

$T(s)$ —message start time, the time required initiating the message passing.

$T(w)$ --the transfer time per word.[14]

So the communication time to pass a message of size L , $T=T(s) + T(w)*L$. $T(w)$ is a certain parameter in a machine.

3.2.3 How to map the task to processors

The most important concern about mapping the tasks is the balance between idle time and communication. It is a conflicting objective: minimizing one increases the other. If we assign all the work to one process, that minimizes the communication between processors, but will cause significant idle time, on the other hand, if we minimize the serialization (idle time), that will introduce communication.

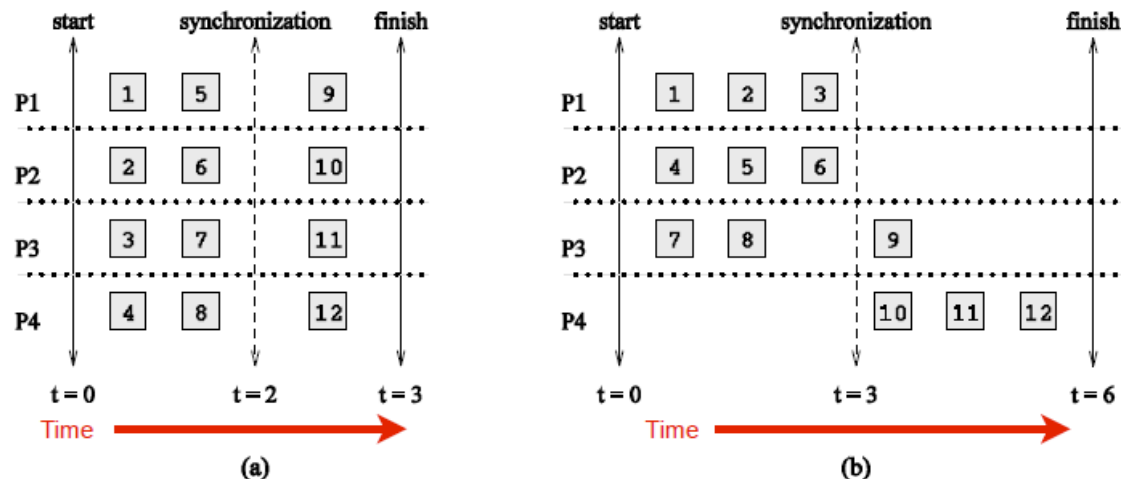


Figure 16. Balance between idle time and communication.

(Source: [25])

As shown in figure 16b, there are 12 serial tasks to complete, and one task can not be compute until the one before it has already been done. In diagram b, processor 1 load task 1, 2, 3 as time goes, like a serial task, processor 2 and 3 are doing the same job. Compared with figure a, the four processors are assigned to the same amount of data, that saves half of the whole computation time.

When the number of task is static, we can set a structured communication, there is constant computation, variable computation time per task. We can agglomerate tasks to minimize the communication, each processor create one task.

For example, consider the following program:

Processor A	Processor B
A1: Read the value X	B1: Read the value X
A2: ADD 1 to data X	B2 : ADD 1 to data X
A3: Store X back to memory.	B3: Store X back to memory.

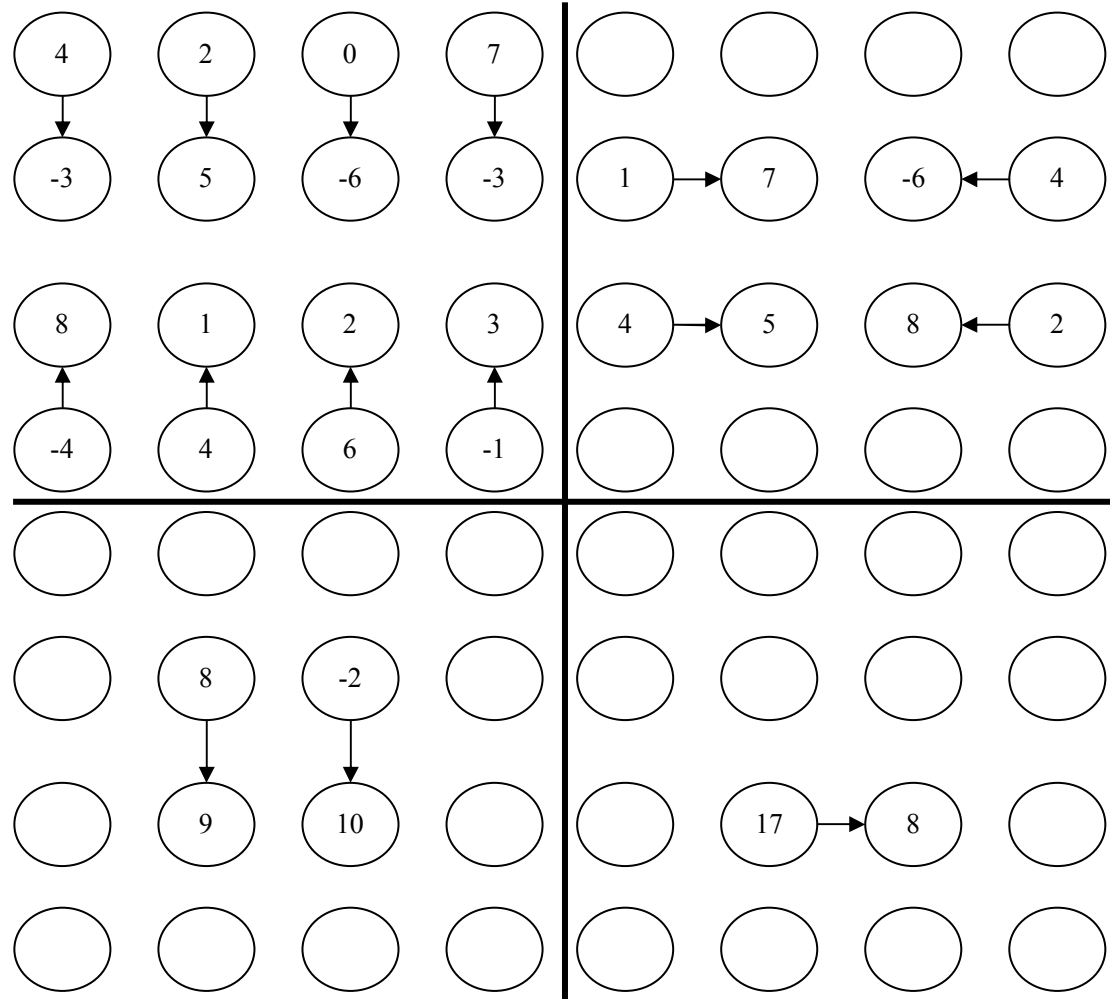
If the instruction B1 is executed between A1 and A3, or instruction A1 is executed between 1B and 3B, the program will produce error data. To solve this problem, we can simply just add one lock to the value X, when processor A

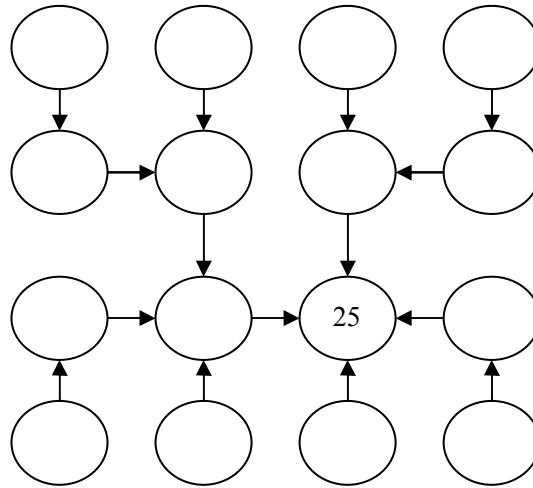
is accessing the data, then the data is locked, that means processor B can't access to the data X. This method can fix the problem, but at the same time, it will slow down the computation.

3.2.4 Case study

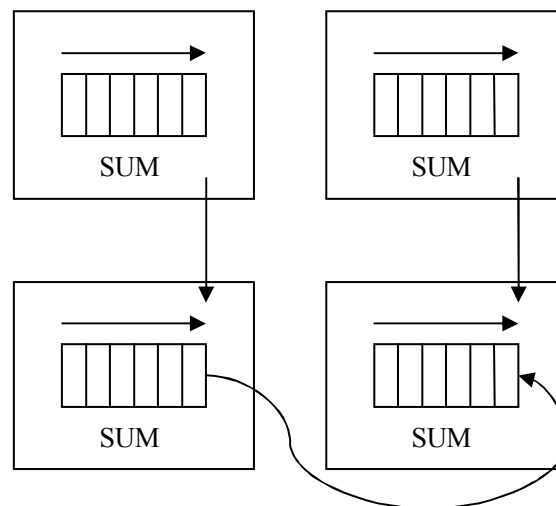
In this section, I will show the study of one simple case, how the whole algorithm was came up.

The example is finding the global sum. [24]





As illustrated by the diagram, the calculation of the global sum can be separated into the “Binomial Tree”.



In the agglomeration part, the whole sum up can be separate into four parts. Every part is responsible for their part of computation, so the four parts can compute concurrently

The sequential execution time can be expressed as:

- χ – time to update element
- n – number of elements
- m – number of iterations
- Sequential execution time is : $m \cdot n \cdot \chi$

The Parallel Execution Time can be expressed as:

- p – number of processors
- λ – message latency
- Parallel execution time $m([\chi n/p] + 2\lambda)$

(Source:[23])

3.3 Software and target multi-core device

The software we are using is ARM workbench IDE v4.0, the integrated ARM Profiler, which is part of RVDS 4.1 Professional (Realview Design suit), is a unique product enabling non-intrusive analysis of embedded software performance for virtually unlimited periods of time. It is capable of running at operational frequencies of up to 400 MHz and can gather profile information covering minutes, hours or days. [21]

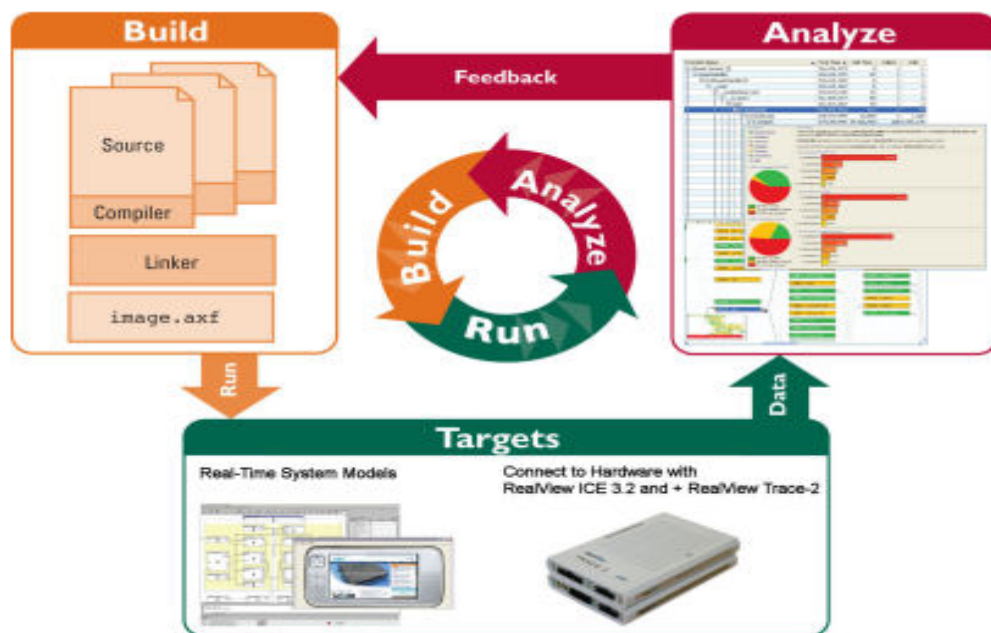


Figure15. ARM Profiler Non-Intrusive Performance Analysis
(Source:[21])

- Support for the latest Cortex™ -A9, Cortex™ -A5, and Cortex™ -M4 processors.
- The highly optimizing ARM Compiler, supporting all modern versions of the ARM, Thumb and Thumb-2 instruction sets and supporting the NEON SIMD instruction set with the vector zing NEON compiler.
- High performance software development using Real-Time System Models of complete ARM processor cores and boards running at around 200MHz.
- Link-time code generation, which enables cross source-file optimization while neatly fitting into existing makes file technology. [21]

The simulation target is Cortex-A9MP, Available as either a single core or configurable multi-core processor, with both synthesizable and hard-macro implementations available.

The ARM Cortex-A9 processor delivers exceptional capabilities for less power than consumed by high performance compute platforms, including

Cortex-A9	
Architecture	ARMv7-A Cortex
Dhrystone Performance	2.50 DMIPS/MHz per core
Multicore	1-4 cores Single core version also available

Figure16. Cortex-A9 Key Features

(Source: 21)

3.4 Algorithms selection

In this project, my aim is first to Study some different exiting algorithms, how they divide one large task into small tasks, how to allow execution of maximum parallel tasks. First we should find several typical algorithms that can represent different classes of algorithms like sorting, searching and signal processing. Secondly, the algorithms should easily reveal the catachrestic of parallel computing, to fulfill the aim of this project. So the selection of algorithms is critical

As mentioned in [24], the work of Phil Colella, who identified seven numerical methods [Colella 2004] called “seven dwarfs”, these seven classes of algorithms was categorized by similarity in computation and data movement. The dwarfs are specified in a high level of abstraction, and showed the communication patterns across a range of applications.

As I am searching suitable algorithms to study on, the classes of these dwarfs are very helpful.

A single dwarf can expand to cover such a disparate variety of methods that it should be viewed as multiple distinct dwarfs.[24]

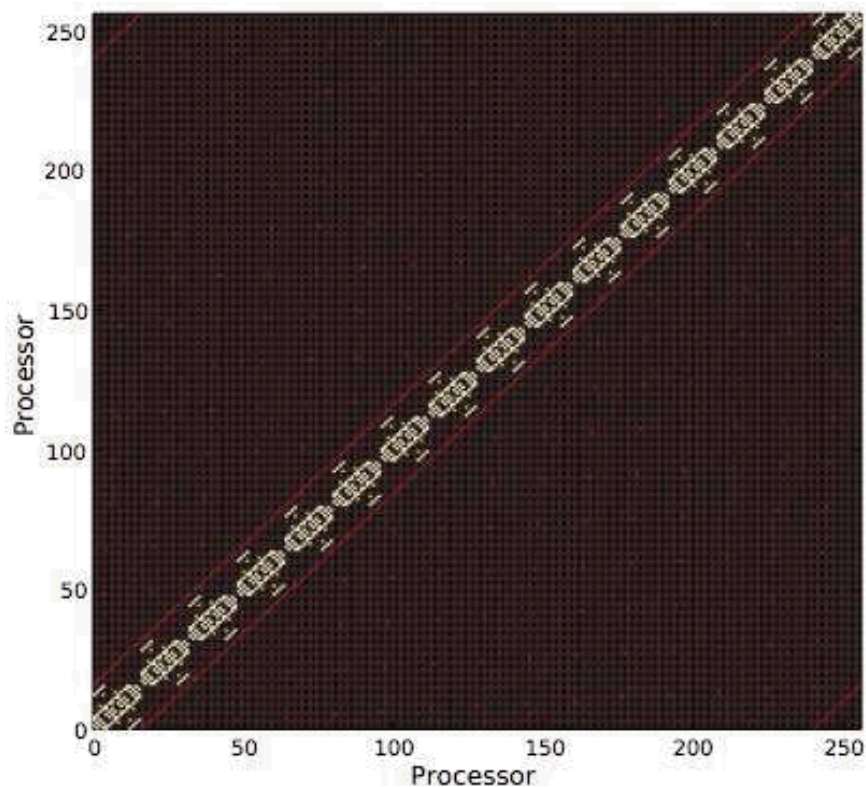


Figure19. The communication pattern of Dense Linear Algebra
(Source: [24])

The first one is dense linear algebra, as we can see in the figure, generally, such applications use unit-stride memory accesses to read data from rows, and stride accesses to read data from columns.[24]

When parallelize this class of algorithms, there will be significant communications between processors. There will be a lot of idle time.

The third dwarf, “Spectral Methods”, Data are in the frequency domain, as opposed to time or spatial domains.

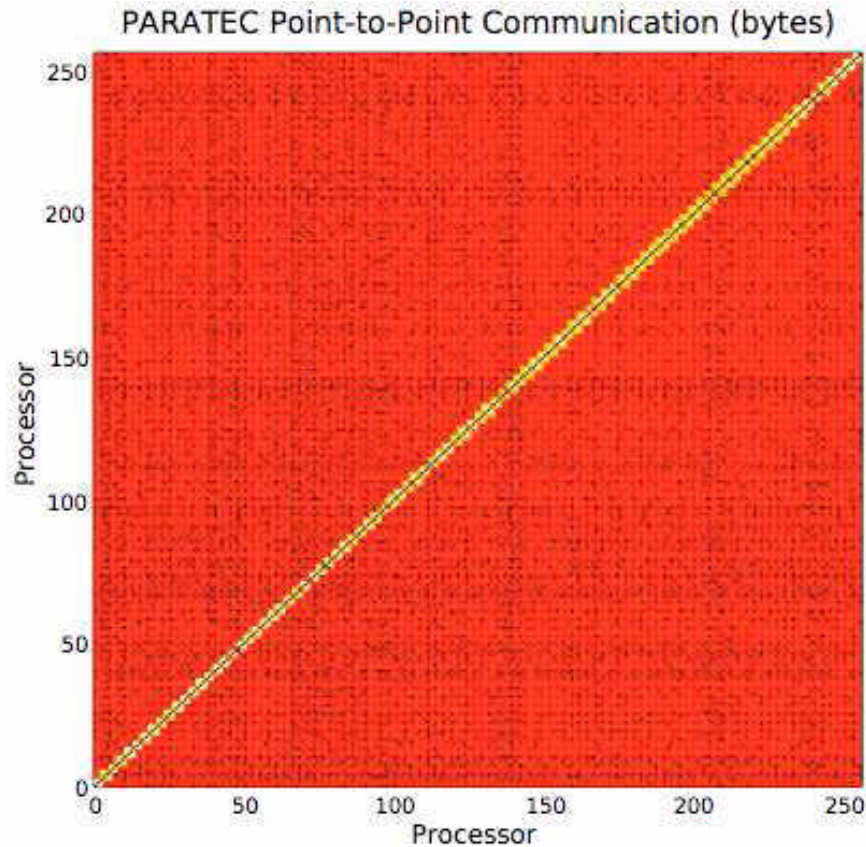


Figure18. The communication pattern of FFT
(Source:[24])

In figure19 is the communication pattern of dense matrices or vectors. The red spot means the 3D FFT requires an all-to-all communication to implement a 3D transpose, which requires communication between every link. [24] We can separate the pattern into parts to analysis. As we can see in the figure, compared with dense linear algebra, it saves a lot of communication time, it only need to communicate when calculating butterfly pattern (will be discussed in the next section).

So after consider the communication patter, the potential of the different algorithms can be parallelized, I chose sequential sum, bubble sort and FFT algorithms to do further study.

3.5 Sequential sum

The general linear first-order recurrence can be expressed as the evaluation of the sequence x_j from the recurrence relation.

$$x_j = a_j x_{j-1} + d_j \quad (j=1, 2, 3, \dots, n, x_0, a_1, a_2, \dots, a_n, d_1, d_2, \dots, d_n \text{ are given})$$



Figure17. Sequential sum computation data flow.

The vertical axis is time vector, the horizontal axis is processing element. The arrow indicated the routing of data. [20]

The operations on the same horizontal level can be parallelized, as we can see in the figure, on each horizontal level only one computation can be performed, that means the parallelism is 1. Each computation requires (except the first one) a routing of one unit to the right at each time level.

In the experiment, I set up one comparison between one thread (serial) and eight threads. Because the software RVDS 4.1 can't profile two processors at one time, so I profiled one CPU core, and by observing this one CPU, I can get the full information about parallel this algorithm,

Serial (one thread):

N=8, it takes 30559 cycles to finish computing.

Overall Analysis	
Processor:	Cortex-A9MP @ ~0 MHz
Date:	22 September 2010
Time:	19:50:29
Instructions:	21,459
Avg Ins/Sec:	214,590
Cycles:	30,559 (estimated)

Parallel (observe one CPU core):

In parallel, when one core finished computing the sum on one data, it will send one message to a "message box", and before the other cores start computing, it will first see if the message box is empty (by checking the flag if it's 0), then use the data to continue calculate.

Overall Analysis	
Processor:	Cortex-A9MP @ ~0 MHz
Date:	22 September 2010
Time:	20:49:30
Instructions:	3,609
Avg Ins/Sec:	36,090
Cycles:	5,254 (estimated)

As we can see in the result, it still takes 5254 cycles for one core to finish computing, and sending messages. 5254 times 8, it takes more cycles than one core, so it takes more time to parallel this sequential sum in ARM processors.

3.6 Bubble sort

Bubble sort is a sort algorithm, works by repeatedly going through the list comparing each pair of unsorted numbers and swapping them if they are in the wrong order.

Bubble sort has worst-case and average complexity both $O(n^2)$, where n is the number of items being sorted. There exist many sorting algorithms with substantially better worst-case or average complexity of $O(n \log n)$. In the practical sorting problems, bubble sort may introduce extra complexity, so bubble sort is not a practical sorting algorithm when n is large.

First sort:

(**3**, **4**, 1, 2, 9) \rightarrow (**3**, **4**, 1, 2, 9) algorithm compares the first two elements, 4 is larger than 3, so don't do anything

(3, **4**, **1**, 2, 9) \rightarrow (3, **1**, **4**, 2, 9), Swap since $4 > 1$

(3, **1**, **4**, **2**, 9) \rightarrow (3, **1**, **2**, **4**, 9), Swap since $4 > 2$

(3, 1, 2, **4**, **9**) \rightarrow (3, 1, 2, **4**, **9**), no swap since $4 < 9$

Second sort:

(**3**, **1**, 2, 4, 9) \rightarrow (**1**, **3**, 2, 4, 9)

(1, **3**, **2**, 4, 9) \rightarrow (1, **2**, **3**, 4, 9)

(1, 2, **3**, **4**, 9) \rightarrow (1, 2, **3**, **4**, 9)

(1, 2, 3, **4**, **9**) \rightarrow (1, 2, 3, **4**, **9**)

Now, the array is already sorted, but the bubble sort algorithm does not aware that it is completed. The algorithm still needs one pass, but without any swap.

Third sort:

(**1**, **2**, 3, 4, 9) \rightarrow (**1**, **2**, 3, 4, 9)

(1, **2**, **3**, 4, 9) \rightarrow (1, **2**, **3**, 4, 9)

(1, 2, **3**, **4**, 9) \rightarrow (1, 2, **3**, **4**, 9)

(1, 2, 3, **4**, **9**) \rightarrow (1, 2, 3, **4**, **9**)

Finally, the array is sorted, and the algorithm can terminate.

In this experiment, I also set one comparison, first use standard bubble sort to sort these 5 numbers, in serial.

Serial sort:

Overall Analysis	
Processor:	Cortex-A9MP @ ~0 MHz
Date:	22 September 2010
Time:	22:16:16
Instructions:	4,044
Avg Ins/Sec:	40,440
Cycles:	5,879 (estimated)

Then in parallel, I suppose there are three CPU cores to do the sorting, one CPU core do the first sort, and then pass the results to the second CPU, when the second CPU finished computing, the third core got its results.

Overall Analysis	
Processor:	Cortex-A9MP @ ~0 MHz
Date:	22 September 2010
Time:	22:17:44
Instructions:	3,734
Avg Ins/Sec:	37,340
Cycles:	5,436 (estimated)

As we can see the result is not good, it only saves 400 cycles in one sort, so the bubble sort algorithm is not suitable to parallelize on ARM processors.

3.7 FFT

3.7.1 Principles of FFT

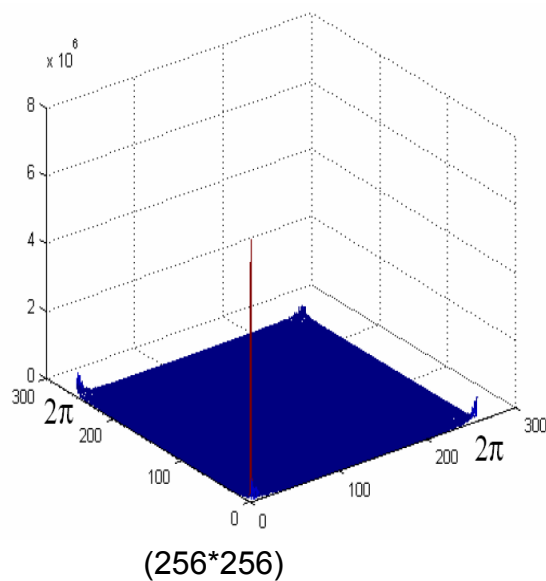


Figure18. FFT can reconstruct the picture with less data
(Source: 28)

The original picture and video needs large amount of data to rebuild, but using FFT, it needs much less data to reconstruct the picture and video.

As we discussed above, there is certain limitations in parallel programming. Because of the specialty of specific problems, sometimes the problem just can't be parallelized. Signal processing is one of the problems that can be parallelized, and FFTs algorithms are of great importance to signal processing. The Fast Fourier Transform (FFT) is a class of algorithms capable of computing the Discrete Fourier Transform (DFT) in $O(n \log n)$ time.

It is widely used in many fields of engineering and mathematics, such as signal processing and polynomial multiplication. FFT shows the relationship between a signal in the time domain and its representation in the frequency domain.

By far the most common FFT is the Cooley–Tukey algorithm.[15], they reduced the complexity from $O(n^2)$ to $O(n \log_2 n)$. FFT on shared-memory architecture is analyzed by A. Norton and A. J. Silberger.(1987) [18].Cui-xiang[17] reduced the complexity of FFT to $O((n \log_2 n) / p)$, p is the number of processors. The basic idea of FFT algorithm in parallel programming is, assume p is the number of processors, $p = 2^{X-Y}$, $N = 2^X$. The FFT consists of a bit-reversal permutation followed by X computational stages. [16]

After the transforming, the first Y parts can be divided into P parts, and map into P separate process elements, each of them have the dataflow of an FFT on 2^Y points. [18] The left part of the work can be subdivided as did before. In this algorithm, each processor will compute $M/4P$ parts of the whole work. The local memory only store the FFT data, and all the data are stored in the global shared-memory. So each processor can compute the data in its only local memory, repeat all the work until each processor did N/P part of the work.

After A. Norton and A. J. Silberger did their analysis, there are a lot of new FFT algorithms have been proposed. Like in [16], in the first stage, FFT is divided by $\log_2(N/p)$ stages, the $N/2P$ butterfly computation is executed, during this stage of computation, no data exchange is needed.

$$x_k = \sum_{n=0}^{N/2-1} x_{2n} w^{-2nk} + w^{-k} \sum_{n=0}^{N/2-1} x_{2n+1} w^{-2nk}$$

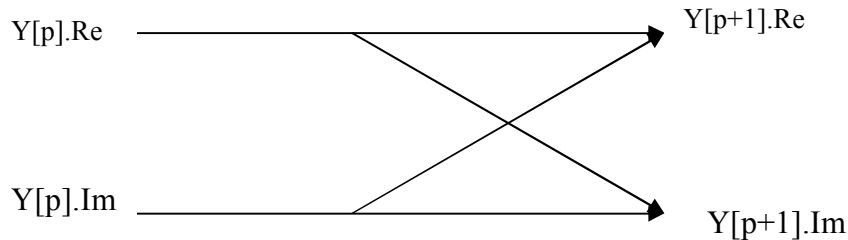
Where $w = e^{2\pi\sqrt{-1}/N}$

As we can see in the code of serial FFT algorithm,

$y[p].\text{Re} = x[p].\text{Re} + W.\text{Re} * x[q].\text{Re} - x[q].\text{Im} * W.\text{Im};$ (1)

$y[p].\text{Im} = x[p].\text{Im} + W.\text{Im} * x[q].\text{Re} + x[q].\text{Im} * W.\text{Re};$ (2)

Form the main data flow pattern of the algorithm. This pattern, often called butterfly,



3.7.2 Implement FFT in serial

In the first experiment, I first use the serial instructions to compute the FFT.

```

angle = -PI * 2 * s * pow((float)2, ((log((float)N) / log((float)2)) - m - 1)) / N;
y[p].Re = x[p].Re + W.Re * x[q].Re - x[q].Im * W.Im;
y[p].Im = x[p].Im + W.Im * x[q].Re + x[q].Im * W.Re;
y[q].Re = x[p].Re - (W.Re * x[q].Re - x[q].Im * W.Im);
y[q].Im = x[p].Im - (W.Im * x[q].Re + x[q].Im * W.Re);
  
```

As we can see in the butterfly pattern, we need to compute multiplication of complex number two times, and addition of complex number two times, but there are still duplicated computation.

1 . Compute $a = Y[p].\text{Im} * W$

2 . Compute $b = Y[p].\text{Im} + a$

3 . Compute $c = Y[p].\text{Im} - a$

When doing the multiplication of complex number, one multiplication can be saved, but the times of addition of complex number is not increasing. That means every time, the butterfly computation will reduce one multiplication of complex number.

The overall analysis is showed in figure 20, the cycles is 329918. Because it's a serial instruction, the main thread is the main function (figure 21). The output is in figure 22, the output is the result of Fourier transform. The simulation only target on the CPU0, which means all the computations were done by one of the CPUs.

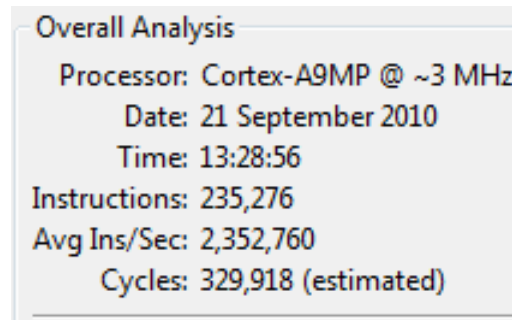


Figure20

```
Attaching the profiler trace to component RTSM_EB_Cortex_A9_MPx2.coretile.core.cpu0.
Simulation is started
X[0]=(28)+j(0)
X[1]=(-4)+j(9.65685)
X[2]=(-4)+j(4)
X[3]=(-4)+j(1.65685)
X[4]=(-4)+j(0)
X[5]=(-4)+j(-1.65685)
X[6]=(-4)+j(-4)
X[7]=(-4)+j(-9.65685)
Simulation is terminating. Reason: Simulation stopped
```

Figure 22

In the experiments, I set eight complex-numbers, from 1-8, the real part is assigned form 1-8, the imaginary parts are all 0.

3.7.3 Theoretical timing analysis

$$x_k = \sum_{n=0}^{N/2-1} x_{2n} w^{-2nk} + w^{-k} \sum_{n=0}^{N/2-1} x_{2n+1} + w^{-2nk}$$

Where $w = e^{2\pi\sqrt{-1}/N}$

From the definition, compare the serial and parallel FFT computation:

T_s : The time serial FFT need to finish computing

T_p : The time parallel FFT need to finish computing

T: The time to calculate one data

N: The number of data to be processed

P: The number of process element

T_{start} : The time to start one thread.

T_{store} : The time to store one data in the shared memory.

T_{commu} : The time on the communication between threads.

So the

$$T_s = T \times N \times \log_2 N \quad (1)$$

In the parallel FFT, there are two phase of computation:

$$T1 = T \times N \times \log_2 P / P \quad (2)$$

$$T2 = T \times N \times (\log_2 N - \log_2 P) / P \quad (3)$$

$$T_{commu} = P \times T_{start} + N \times T_{store} \times \log_2 P \quad (4)$$

So, all the time it consumed is:

$$\begin{aligned} T_p &= T1 + T2 + T_{commu} \\ &= T \times N \times \log_2 P / P + T \times N \times (\log_2 N - \log_2 P) / P + P \times T_{start} + N \times T_{store} \times \log_2 P \\ &= T \times N \times \log_2(N) / P + P \times T_{start} + N \times T_{store} \times \log_2 P \end{aligned} \quad (5)$$

So the speedup is the, as explained in section 2.1, T_p divided by T_s :

$$T_p / T_s = (T \times N \times \log_2 N) / (T \times N \times \log_2(N) / P + P \times T_{start} + N \times T_{store} \times \log_2 P) \quad (6)$$

According to formula 6, we can speed up the calculation by increasing the number of processor P, and minimize T_{start} and T_{store} .

3.7.4 Parallelize FFT using message passing approach

As FFT algorithm is a divide-and-conquer algorithm, so it's very suitable to develop parallel computing algorithm.

When n=8, the FFT computation flow is in figure 23.

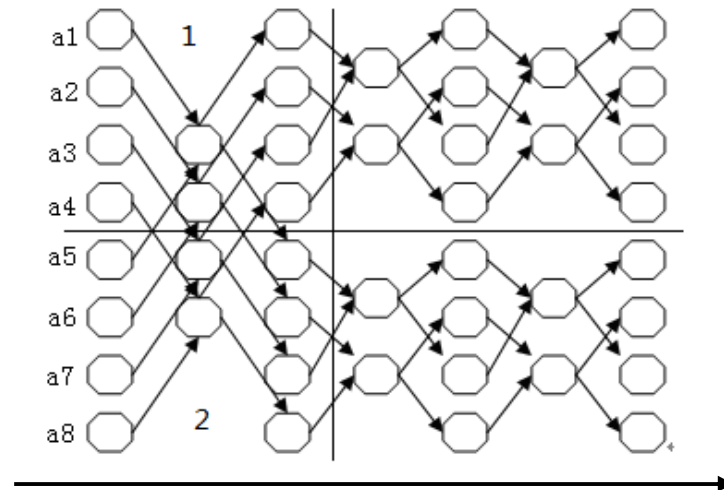


Figure23. FFT computation flow (n=8)

By study different kinds of parallel FFT algorithms, I chose to use the algorithm proposed by Jun Ho Bahn, Jungsook Yang and Nader Bagherzadeh [16] in 2008. Because it separates the FFT into two parts, the odd group and even group, that suits this project. On the other hand, some parallel FFT algorithms focused on saving the cycle time, some focused on by passing the cache, saving the memory access. The algorithms I chose is the balanced work load and minimized communication overhead lead to fast operation of FFT. By using a cycle-accurate simulation and complexity analysis, they showed that the algorithms outperform other parallel FFT algorithms with similar specifications.

- 1) Data balance: All processors should be assigned to an equal amount of work. In particular, sequential parts should be limited since they will be slow down the parallelism
- 2) Synchronization Overhead: Synchronization should involve as little overhead as possible and threads should not wait unnecessarily at synchronization points.
- 3) Avoiding incorrect sharing: one process shouldn't be able to change the data of another process's memory, which is a very dangerous behavior. [26]

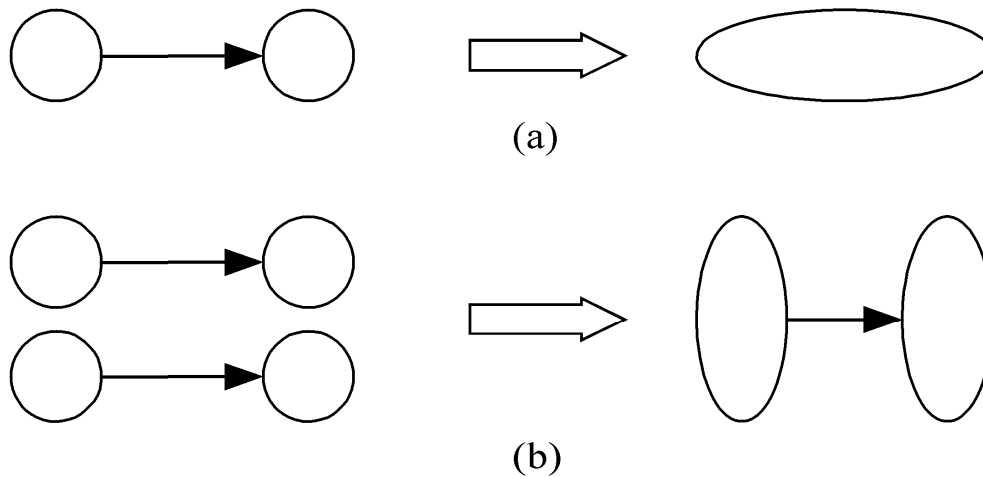


Figure28. Combine groups of sending and receiving tasks

When two tasks are sending to one processor, they can be combined as one. In this project, I set the number of processors to be 2, because the Cortex A9-MPx2 processor can only configured two processors. So the whole sequence will be divided into two parts.

As shown in the figure 23, the upper 4 numbers are grouped together, and the other 4 are one group, in the phase 1, 2, the two groups need to communicate with each other, there are $\log_2(n) - 1 - \log_2(m) - 1$ steps in phase 1. But in the shared memory system, the two threads don't need to send or receive messages from the other, because they share the global memory. So when one thread has finished the computation, it just need send one signal (or change the value of one flag) to the other thread, so the second thread can continue to compute, using the data in the global memory. In this way, the FFT can be done in a shorter time.

Parallelize FFT using message passing approach

```

1: Input:  $a = (a_0, a_1, a_2, a_3, \dots, a_N)$ 
2:  $j = 2$ 
3: for  $e \leftarrow 0$  to 1 do
4:    $t = 2e, l = 2^{(e + \log_2(N/2))}, q = n/2l, z = w^q, j = j-1, v = 2j$ 
5:   for  $i \leftarrow 0$  to 1 do
6:     if  $(i \bmod t = i \bmod 2t)$  then
7:       receive data block from Core1
8:       for  $k \leftarrow 0$  to  $N/2 - 1$  do
9:          $m = (i * N/2 + k) \bmod l$ 
10:         $a[k] = a[k] + a[k + N/v] * z^m$ 
    
```

```

11:       $a[k + N/v] = a[k] - a[k + N/v] * z^m$ 
12:  end for
13:      Send transformed data to Core 0
14:  end if
15: end for
16: end for

```

In this algorithm, I set “receive” and “send “. When I am doing the message passing approach, one problem accrued, where should the sender store the messages? First the message can’t be stored in the sender’s memory, because the sender which is a process may finish before the message is arrived and received, if the process is finished, the memory may be cleared by other process. Second the message can’t be stored in the receiver’s memory, because that means one process can change the data of another process’s memory, that is a very dangerous behavior.

So the message should be stored in a “pickup points”, I did in this project is to create a message box, the senders will check if the receiver have a message waiting, if not , the sender will deposit the message, and notify the receiver that there is a message waiting, finally the receiver can get the message successfully.

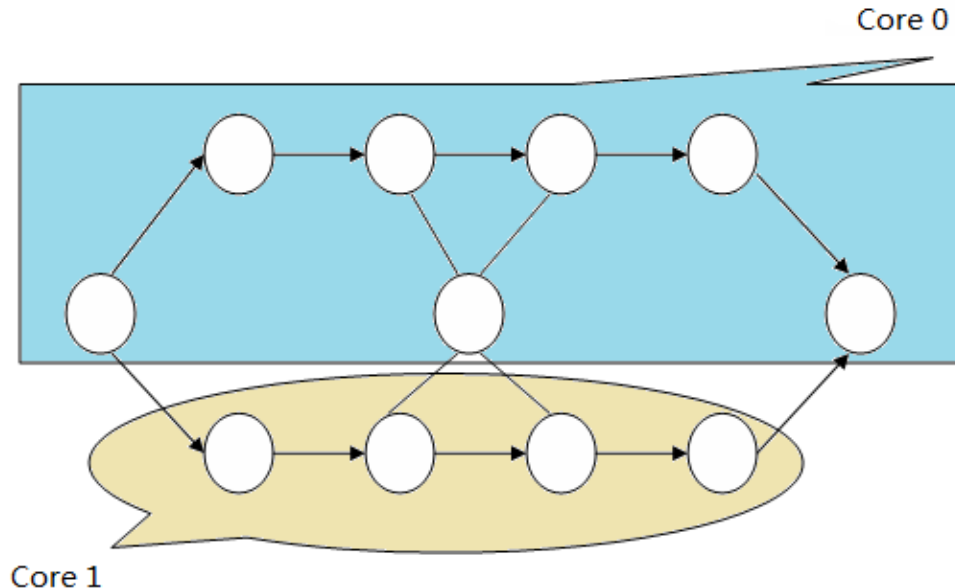


Figure24. Task separation

As showed in figure 24, because the software RVDS 4.1 can’t profile two processors, so I just profile the CPU 0. The result is showed in figure 25 and 26. But as we discussed before, the two cores are basically doing the same amount of job, and if Core 1 hasn’t finished computing, Core0 won’t get the

right results. So, we can anticipate the work of Core1 by analysis the work of Core0.

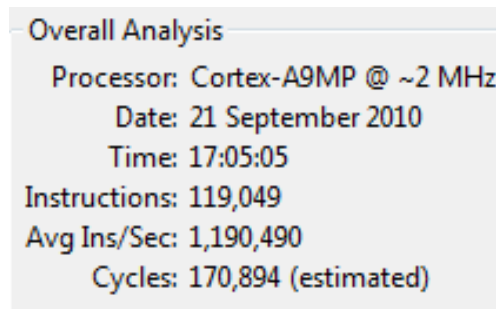


Figure25 Overall analysis of Core 0

It takes 170,894 cycles to finish computing, theoretically, it should save about half cycles to compute the algorithms, but when actually doing the task, the intercommunication is time consuming,

As we can see in the figures, parallel FFT is more efficient than the serial FFT. Because the two processors can work concurrently, and at the second stage the two groups don't need to communicate with each other.

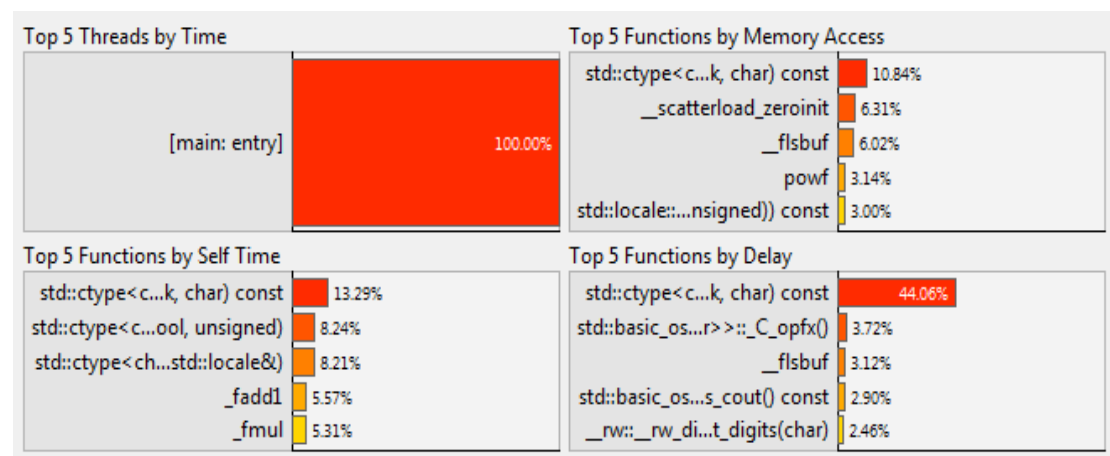


Figure 26

Figure 26 shows the top5 threads by time, top5 functions by memory access, top5 functions by self time and top 5 functions by delay. As we can see the thread is main function, because there is only one thread in one processor. The thing we should notice is the top 5 functions by delay

```
Simulation is started
X[0]=(28)+j (0)
X[1]=(-4)+j (9.65685)
X[2]=(-4)+j (4)
X[3]=(-4)+j (1.65685)
X[4]=(-4)+j (0)
X[5]=(-4)+j (-1.65685)
X[6]=(-4)+j (-4)
X[7]=(-4)+j (-9.65685)
Simulation is terminating. Reason: Simulation stopped
```

Figure 27

Figure 27 shows the result is right, means the function is right and the message passing mechanism is working.

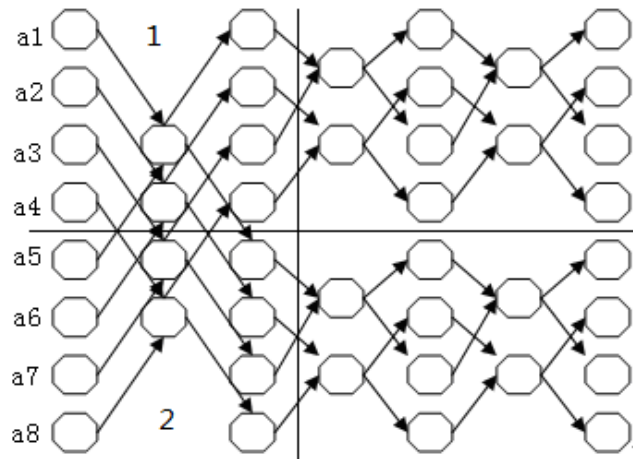
Theoretically, it should save about half cycles to compute the algorithms, but when actually doing the task, the intercommunication is time consuming. As we can see in the figures, parallel FFT is more efficient than the serial FFT. Because the two processors can work concurrently, and at the second stage the two groups don't need to communicate with each other.

3.7.4 Parallelize FFT using shared memory approach

A shared memory system is relatively easy to program since all processors share a single view of data and the communication between processors can be as fast as memory accesses to a same location, a method of conserving memory space by directing accesses to what would ordinarily be copies of a piece of data to a single instance instead.

I first set a shared memory space, this global memory can be accessed by both processors, when doing the FFT calculation, firstly, core0 read and load the data in the shared space, and calculate, in phase one, when core0 has calculated the data which core1 need (means the data need to share with core1), core0 will copy the data to the shared place, so core1 can read the data and continue its computing.

As we can see in this algorithm, in shared memory, it is convenient to save the data in global memory, and it doesn't need to send the message to other processor, and doesn't need to check the message box. Before the FFT function begins to calculate, it will first read the data from global memory, and when finished, it will save the data back. It seems saving a lot of time, but when the simulation tool actuarially implement the algorithms, the other processor always tried to access the memory when the memory is locked. In the original algorithms of Jun Ho Bahn, Jungsook Yang and Nader Bagherzadeh [16], there is no lock and unlock mechanisms, so I put a lot of attention on putting the lock and unlock.



The figure showed the data flow of FFT, as we can see the communication between the two groups, basically all happened in phase1, so I set the 'lock' before the every communication point, and 'unlock' after it. So every communication point will be protected, all the data need to share with the two processors can't be accessed by both processors.

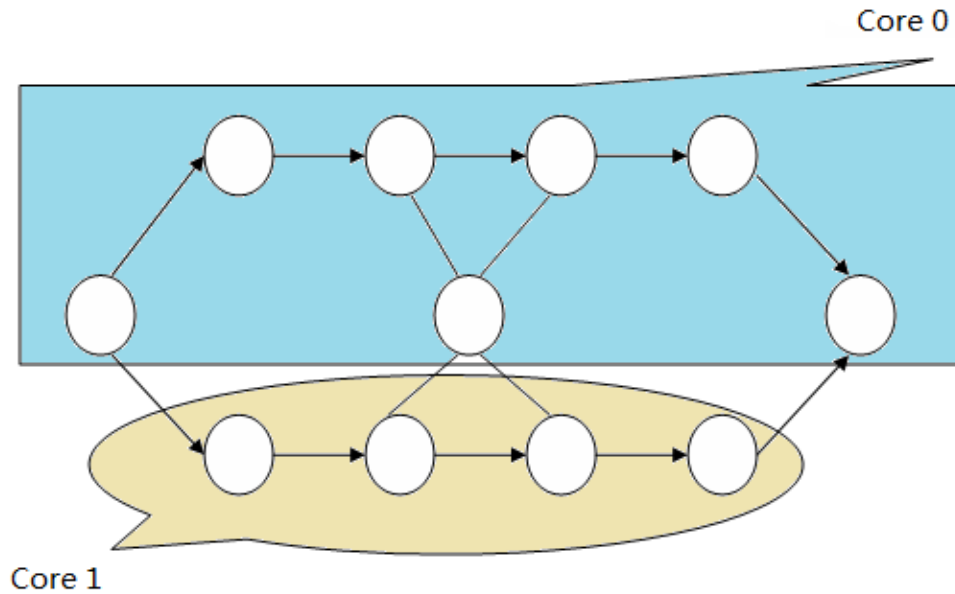
Parallelize FFT using shared memory approach:

```

1: Input:  $a = (a_0, a_1, a_2, a_3, \dots, a_N)$ 
2:  $j = 2$ 
3: for  $e \leftarrow 0$  to 1 do
4:    $t = 2^e, l = 2^{(e + \log_2(N/2))}, q = n/2l, z = w^q, j = j-1, v = 2j$ 
5:   for  $i \leftarrow 0$  to 1 do
6:     if  $(i \bmod t = i \bmod 2t)$  then
7:       read data from shared memory, lock
8:       if (the data isn't valid, read flag==0)
9:         continue;
10:      else
11:        for  $k \leftarrow 0$  to  $N/2 - 1$  do
12:           $m = (i * N/2 + k) \bmod l$ 
13:           $a[k] = a[k] + a[k + N/v] * z^m$ 
14:           $a[k + N/v] = a[k] - a[k + N/v] * z^m$ 
15:        end for
16:      store data in shared memory place, unlock
17:    end if
18:  end for
19: end for

```

In this algorithm, the 'lock' and 'unlock' mechanisms ensured the data can only be accessed by one processor.



The figure showed the separation of task. The two processors divide the entire task into two parts. As we can see in the diagram, Core0 and Core1 are basically doing the same job.

Core0 and Core1 start the simulation at the same time, when the communication point comes, and the two cores will exchange data information (using message passing or shared memory). If one of them hasn't finished computing, or the communication data wasn't ready, or there is a read miss occurred, one of the processors will do idle cycle, wait for the data to be ready. So when I am simulating the computation, if Core0 has finished computing, the whole simulation will be stopped.

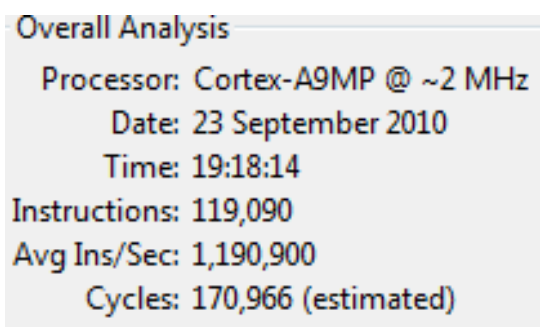


Figure27. Overall analysis of core0

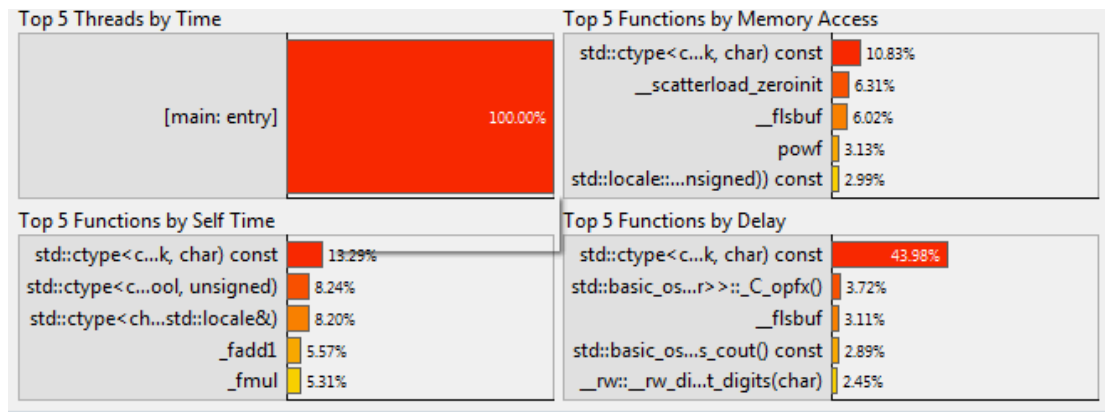


Figure28

In figure28, compared with the analysis file of message passing, the main thread is also the main function, but the function by delay is the read miss, the main obstacle is when one Core0 was doing the computing, I set a lock, that means Core1 can't access the data in the global memory, when Core1 can't read the data from shared memory, the instruction 'continue' will cause one idle cup cycles.

```
Simulation is started
X[0]=(28)+j(0)
X[1]=(-4)+j(9.65685)
X[2]=(-4)+j(4)
X[3]=(-4)+j(1.65685)
X[4]=(-4)+j(0)
X[5]=(-4)+j(-1.65685)
X[6]=(-4)+j(-4)
X[7]=(-4)+j(-9.65685)
Simulation is terminating. Reason: Simulation stopped
```

Figure 29

The result is the same, means the data exchange, and the whole function is right.

As we can see in the analysis files, one of the big trade-offs between the two programming patterns is that it is easier to write a working version of a program in shared memory. Like in this experiment, in shared memory approach, when core0 has something that core1 need, it just needs to store the data in one shared places.

However, it is easier to optimize programs under message passing, since it is easy to see where communication occurs in the program. When using the shared memory approach, core1 may stop computing (the continue instruction), as the data shared is not ready, or the data is not valid, core1 can't access the value.

Chapter 4: Conclusion

In this project, first I have discussed the several algorithms, including the numerical and sorting algorithms. Secondly, I used the t Jun Ho Bahn, Jungsook Yang and Nader Bagherzadeh [16] algorithms to modified traditional serial computing to the parallel algorithm. Thirdly, I used two methods to make the use of it on ARM processors.

As we can see through the discussion, the performance of a computer program or algorithm to be inversely proportional to the CPU cycles consumed during the execution of the program. The algorithms are used to solve a problem and on the skill with which the algorithm is implemented on the computer. So the algorithms must cooperate with the architecture of the computer, which means on some certain architecture, some algorithms are not suitable to parallelize.

At any stage within an algorithm, the parallelism of the algorithm is defined as the number of arithmetic operations that are independent and can therefore be performed in parallel. Algorithms like sequential sum and bubble sort, the parallelism are 1.

Like the sequential sum and bubble sort, they need multi-cores to compute, but one processor can't perform computing except the earlier one has finished computing. When parallelize them on the ARM processors, all the processors will be waiting for the earlier one to finish computing, the idle time is magnificent, and this means the there is no meaning to parallelize this algorithms on ARM processors.

In chapter 3, we focused on the parallel FFT algorithm. I first came up with the data flow of FFT, and find out that there are two phases, if we separate the 8 numbers FFT into two parts, each processor is responsible for 4 numbers, only in phase one ,then need to exchange data with each other, in phase two, they can compute concurrently without any communication.

There are many different ways to parallelize FFT, some considered more to save the computation time, some focused on the cache technology, tried to bypass the cache. I chose to use the algorithm proposed by Jun Ho Bahn, Jungsook Yang and Nader Bagherzadeh [16] in 2008, The balanced work load and minimized communication overhead lead to fast operation of FFT. By using a cycle-accurate simulation and complexity analysis, they showed that the algorithms outperform other parallel FFT algorithms with similar specifications.

I set two ways to parallelize the FFT, message passing and shared memory. In message passing approach, I set a message box, when the data need to send to the other processor, it will send the message to the message box, the other processor will first check the message box, read the data, and continue computing. As we can see in the analysis files, the core0 is saving nearly half cycles than the serial FFT, because there are communications between the

two processors.

In shared memory approach, I first set a global memory place, which two processors all have the access to it. When the two processor need to exchange data, instead of sending messages, one processor can just save the data in the global memory. As we can see through the analysis files, in shared memory, it takes more cycles than the message passing approach. Because there are read miss in the shared memory, when core1 wants to access the data, but the data is still not ready, core1 has to wait another cycle to wait for it. In serial computation, it takes 329,918 cycles to finish computing.

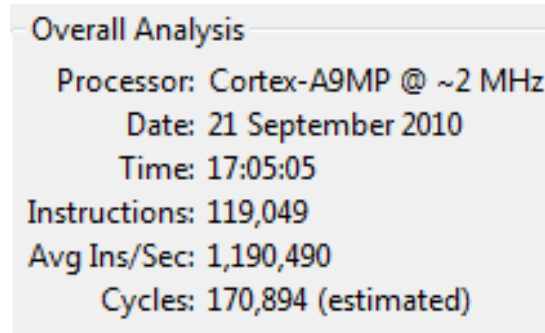
Overall Analysis	
Processor:	Cortex-A9MP @ ~3 MHz
Date:	21 September 2010
Time:	13:28:56
Instructions:	235,276
Avg Ins/Sec:	2,352,760
Cycles:	329,918 (estimated)

When using the shared memory approach, it only takes Core 1 170,966 cycles to finish computing. Because the limitation of the simulation tools (it can only profile one CPU at a time.), we can only anticipate cycles of another core. But as we discussed before, the two cores are basically doing the same amount of job, and if Core 1 hasn't finished computing, Core0 won't get the right results. So, all the work will be finished in 170,966 cycles.

The main obstacles of shared memory is when one Core0 was doing the computing, I set a lock, that means Core1 can't access the data in the global memory, when Core1 can't read the data from shared memory, the instruction 'continue' will cause one idle cup cycles. So I put a lot of effort on where to insert the 'lock' and 'unlock'.

Overall Analysis	
Processor:	Cortex-A9MP @ ~2 MHz
Date:	23 September 2010
Time:	19:18:14
Instructions:	119,090
Avg Ins/Sec:	1,190,900
Cycles:	170,966 (estimated)

In message passing approach, it takes 170,894 cycles to finish computing.



Overall Analysis

Processor:	Cortex-A9MP @ ~2 MHz
Date:	21 September 2010
Time:	17:05:05
Instructions:	119,049
Avg Ins/Sec:	1,190,490
Cycles:	170,894 (estimated)

In compared with shared memory, it is not convenient to send every data as a message to Core1, when using the shared memory, all the data will be stored in the global memory, and it is easy for the other processors to read most of the time. However, the advantage of message passing is that, there will not be lock and unlock mechanisms, that means there will be much less idle cycles. So the main obstacles is dividing the task and mapping them to processors, try to minimize the communication.

Chapter 5: Future Development

In this project, I studied several selected parallel algorithms, and combined them with ARM processors. There are still many ways to improve the parallel computing.

5.1 ARM Cortex-9x Mpcore

In this project, although I am using the cortex-9x processors, but still the architecture is not designed for parallel computing, the entire memory coherency problem must be solved by software program, it is time consuming, and not suitable for practical applications.

As we can see in figure30, is the new architecture of Cortex-9x Mpcore. In order to solve the memory coherency problem, ARM holdings add a snoop control unit (SCU), Memory coherency in a Cortex-A9 MPCore is maintained following a weakly ordered memory consistency model. The function of SCU is to maintain the data coherency of all the Cortex-9x processors. Every CPUs will have its own private timer and watchdogs,

Cache coherency among L1 data caches of the Cortex-A9 processors in the cluster is maintained when the Cortex-A9 processors are operating in Symmetric Multi-Processing (SMP) mode. This mode is controlled by the SMP bit of the Auxiliary Control Register. To be kept coherent, the memory must be marked as Write-Back, Shareable, and Normal memory. [27]

Because there is a hardware channel for the CPUS to communicate, and there are certain instructions to implement the communication, it saves a lot of time, and a lot of effort to write the code.

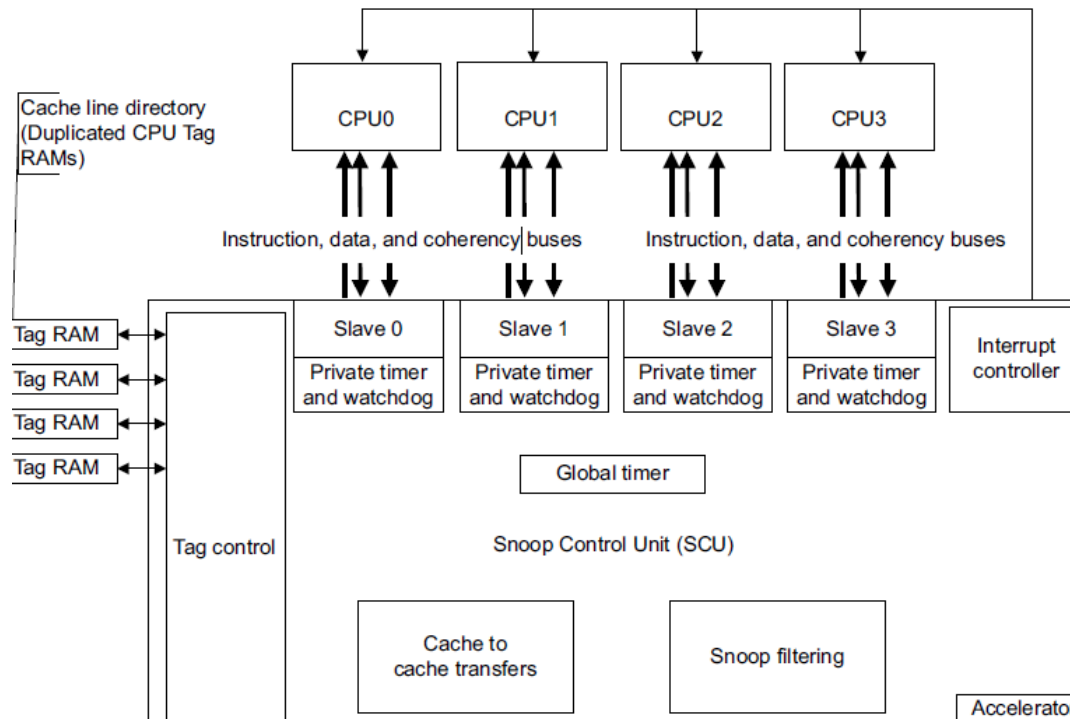


Figure30
(Source: 27)

5.2 Parallelize FFT

In parallelization of FFT, 2D FFT is a new level of parallelism. To parallelize the 2D FFT, after the transforming, the first Y parts can be divided into P parts, and map into P separate process elements, each of them have the dataflow of an FFT on 2^Y points. [18]

The left part of the work can be subdivided as did before. In this algorithm, each processor will compute $M/4P$ parts of the whole work. The local memory only store the FFT data, and all the parts are stored in the global shared-memory. So each processor can compute the data in its only local memory, repeat all the work until each processor did N/P part of the work.

After A. Norton and A. J. Silberberger did their analysis, there are a lot of new FFT algorithms have been proposed. Like in [16], in the first stage, FFT is divided by $\log_2(N/p)$ stages, the $N/2P$ butterfly computation is executed, during this stage of computation, no data exchange is needed.

There will be four butterfly patterns in 2D FFT.

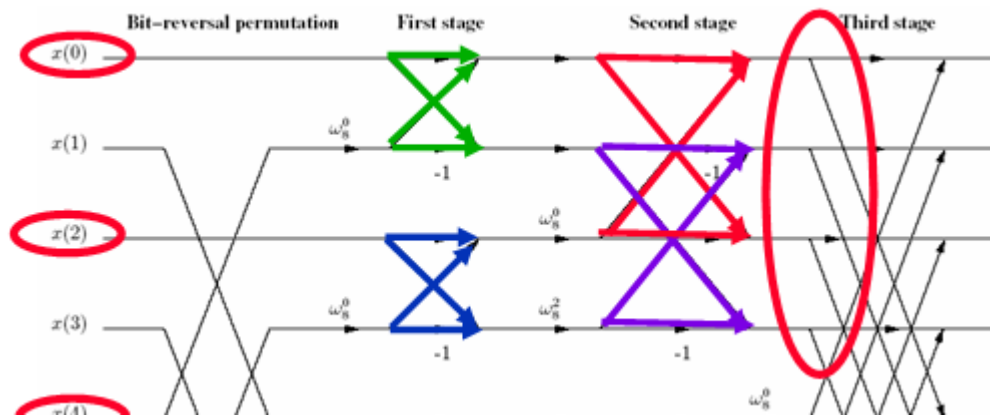


Figure 31 Four butterfly operations
(Source: [30])

Bibliography

1. J. L. Hennessy and D.A. Patterson (2003) 3rd edition "Computer Architecture". Morgan Kaufmann Publishers. Page 529-560
2. A.A.Ahmad (2002) "Object Oriented Parallel Programming" 0-7803-7505-X/02/02002 IEEE
3. A. S. Tanenbaum.(1995)" Distributed Operating Systems." Prentice-Hall, INC. Page 8-12. 289-320
4. Rodgers, David P. (June 1985). "Improvements in multiprocessor system design". *ACM SIGARCH Computer Architecture News archive* (New Yorok, NY, USA: ACM) **13** (3): 225–23
5. Jun Tan, Xingshu Chen "An Optimized Parallel FFT Algorithm on Multiprocessors with Cache Technology in Linux" 978-0-7695-3498-5/08 © 2008 IEEE
6. Tartalja, Igor.(1996) "The cache coherence problem in shared-memory multiprocessors : software solutions". Page 40-48
7. J.B. Carter and J. K. Bennett (1995)"Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems"
8. D. Comer(1984) "Operating system design the XINU approach" Prentice-Hall, INC.pp Page 93-98
9. Amdahl, Gene (1967). "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities" *AFIPS Conference Proceedings* (30): Page 483–485.
10. S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, Dec. 1996.
11. Milo M. K. Martin(2005) "Formal Verification and its Impact on the Snooping versus Directory Protocol Debate" *IEEE Computer*, 2005.
12. P, Silvio ,M, David G.(1993)" Performance and Scalability Aspects of Directory-Based Cache Coherence in Shared Multiprocessors" 1993. ICPP 1993. International Conference on Parallel Processing,
13. Chaiken,D, Fields,C (1990) "Directory-based cache coherence in large-scale multiprocessors" *Computer* Jun 1990
14. Ian Foster (1995)"Designing and Building Parallel Programs" Addison-Wesley, 1995 ISBN 0-201-57594-9
15. James W. Cooley & John W. Tukey (1965): "An algorithm for the machine calculation of complex Fourier series", *Math. Comput.* 19, Page 297–301.
16. Jun H. Bahn, Jungsook Yang and Nader Bagherzadeh(2008) "Parallel FFT Algorithms on Network-on-Chips" 978-0-7695-3099-4/08© 2008 IEEE
17. Z. Cui-xiang, H. Guo-qiang, and H. Ming-he. Some new parallel fast Fourier transform algorithms. In *PDCAT '05*, pages 624–628, Washington, DC, USA, 2005. IEEE Computer Society.
18. A. Norton and A. J. Silberger. Parallelization and performance analysis of the Cooley-Tukey FFT algorithm for shared-memory architectures. *IEEE Trans. Computer.*, 36(5):581–591, 1987.

19. A. Agarwal et al., "An Evaluation of Directory Schemes for Cache Coherence," *Proc. IS'88 Int'l Symp. Computer Architecture*, CS Press, Los Alamitos, Calif. Order No. 861. June 1988, pp. 280-289.
20. R W Hockney, C R Jesshope. *Parallel Computers 2 ARCHITECTURE, PROGRAMMING AND ALGORITHMS*. IOP Publishing LTD 1998
21. ARM holding <http://www.arm.com/products/tools/arm-workbench-ide.php>
22. Filipe Cabecinhas, Nuno Lopes." Parallel Fast Fourier Transform" Instituto Superior T ecnico Av. Rovisco Pais, 1049-001 Lisbon, Portugal
23. Jun Tan, Xingshu Chen, Long Xiao, School of Computer Science Sichuan University Chengdu, China." An Optimized Parallel FFT Algorithm on Multiprocessors with Cache Technology in Linux" 978-0-7695-3498-5/08 \$25.00   2008 IEEE DOI 10.1109/ISCSCCT.2008.252
24. "The Landscape of Parallel Computing Research: A View from Berkeley" Electrical Engineering and Computer Sciences University of California at Berkeley
25. "Principles of Parallel Algorithm Design" Prof. Dr. Cevdet Aykanat Bilkent  niversitesi Bilgisayar M hendisliđi B l m 
26. FFT Program Generation for Shared Memory: SMP and Multicore Franz Franchetti, Yevgen Voronenko, and Markus P uschel Electrical and Computer Engineering Carnegie Mellon University . SC2006 November 2006, Tampa, Florida, USA 0-7695-2700-0/06 \$20.00  c 2006 IEEE
27. Cortex™-A9 MPCore Revision: r2p2 Technical Reference Manual. Copyright   2008-2010 ARM. All rights reserved. ARM DDI 0407F (ID050110)
28. "Optimizing the Fast Fourier Transform on a Multi-core Architecture" Long Chen; Ziang Hu; Junmin Lin; Gao, G.R.; IEEE International Parallel and Distributed Processing Symposium, 2007
29. A. G. and P. I. Parallel implementation of 2-d FFT algorithms on a hypercube. In *Proc. Parallel Computing Action, Workshop ISPRA*, 1990.
30. S. L. Johnsson and R. L. Krawitz. Cooley-tukey FFT on the connection machine. *Parallel Computing*, 18(11):1201– 1221, 1992.
31. K. B. Theobald. *EARTH: An Efficient Architecture for Running Threads*. PhD thesis, May 1999

Appendix: Source code:

```
#include<stdio.h>
#include<string.h>
#include<math.h>
#define PI 3.1415926
char c;
main()
{int N=8;
int i;
float FFT();
float x[100],XK[100];

printf("Input x(n) \n");
for(i=0;i<=N-1;i++)
{printf("x(%d)=",i);
scanf("%d",x+2*i);

}

*XK=FFT(N,x);

for(i=0;i<=N-1;i++)
{printf("x(i)=%f",*(XK+2*i));
if(XK[2*i+1]==0) continue;
if(XK[2*i+1]>0) printf("+j*%f",XK[2*i+1]);
else printf("-j*(%f)",XK[2*i+1]);
}

}

float FFT(int N,float *x)
{int i;
float *XK;
float *XK1,*XK2;
if(N==2)
{XK[0]=x[0]+x[2];
XK[1]=x[1]+x[3];
XK[2]=x[0]-x[2];
XK[3]=x[1]-x[3];
return XK[2*N];
```



```
}
else
{float *x1,*x2;
for(i=0;i<=N-1;i++)
{x1[2*i]=x[4*i];
x1[2*i+1]=x[4*i+1];
x2[2*i]=x[4*i+2];
x2[2*i+1]=x[4*i+3];
}

*XK1=FFT(N/2,x1);
*XK2=FFT(N/2,x2); /

for(i=0;i<=N/2-1;i+=2) /*butter fly A+BC*/
{XK[i]=XK1[i]+XK2[i]*cos(2*PI*i/N)+XK2[i+1]*sin(2*PI*i/N);
XK[i+1]=XK1[i+1]+XK2[i+1]*cos(2*PI*i/N)-XK2[i]*sin(2*PI*i/N);
}

for(i=N/2;i<=N;i++)
{XK[i]=XK1[i]-XK2[i]*cos(2*PI*i/N)-XK2[i+1]*sin(2*PI*i/N);
XK[i+1]=XK1[i+1]-XK2[i+1]*cos(2*PI*i/N)+XK2[i]*sin(2*PI*i/N);
}

return XK[2*N];
}
}
```

```
#include <stdio.h>
void main()
{
    int sum[100];
    int N,i;
    int m;
    printf("input the n\n");
    scanf("%d",&N);
    for(i=0;i<N;i++)
        sum[i]=i;
    for(i=0;i<N;i++)
        printf("%d+%d=\n",sum+i,sum+i+1);
}
```

;;; Copyright ARM Ltd 2001. All rights reserved.

```
AREA    Heap, DATA, NOINIT
EXPORT HeapBottom

EXPORT UserStackSpace
EXPORT PROCESS_STACK_0;

export shared_space
export free_space

; Create dummy variable used to locate bottom of heap

HeapBottom    SPACE    4

;StackUser      DCD    UserStackSpace + (13-1) * 4;// FOR USER
UserStackSpace    SPACE    13 * 4                ;// FOR USER
free_space      space 100*4
shared_space     space 16
PROCESS_STACK_0 ;                                // FOR USER
SPACE    10*100*4;P_STACK_SIZE * MAX_PROC;
END

;//=====SWI Parameters=====
;0x11 create process
;0x22 kill process
;0x00 halt system
;0x33 process yield
;0x44 resume process
;0x55 os init process
;0x66 os handle the abort;
;0x01 call for printhex
;0x02 call for putchar
;0x03 call for putstring

;//0x77 Message passing: Destination process ID is stored in r1
;//                          Message Data is stored in r0
;OS has 30 32-bit space stack
;Each process has 26 space stack
;//=====

AREA Initialise, CODE, READONLY

EXPORT Start
EXPORT Undefined_Handler
```

```
EXPORT SWI_Handler
EXPORT Prefetch_Handler
EXPORT Abort_Handler
```

```
IMPORT putstring
IMPORT printhex
IMPORT putchar
IMPORT get_prime
import PROCESS_MESSAGE_BOX
import free_space
import shared_space
IMPORT Vector_Init
IMPORT TTBBBase
IMPORT StackTop
IMPORT HeapBottom
IMPORT UserStackSpace
IMPORT SvcStackSpace
IMPORT PROCESS_STACK_0
```

```
;//===== Constant Definition =====
```

```
MAX_PROC      EQU 10
PCB_SIZE      EQU 10*4
P_STACK_SIZE  EQU 100*4
P_MSG_SIZE    EQU 1*4
```

```
P_ID          EQU 0
P_STATE       EQU 1*4
P_PC          EQU 2*4
P_SP          EQU 3*4
P_LR          EQU 4*4
P_REG         EQU 5*4
P_PRE         EQU 6*4
P_NEXT        EQU 7*4
P_ENTRY       EQU 8*4
P_MESSAGE     EQU 9*4
```

```
OS_TCB_ENTRY  EQU 0
OS_COUNTER    EQU 4
OS_LAST_PCB   EQU 8
```

```
P_ST_RUN      EQU 1
```

```
ldr sp,StackSvc      ;// SVC_STACK
BL init_os            ;// Operating system initialization
BL init_user
```

```
MOV r0, #0x18 ; angel_SWIreason_ReportException
LDR r1, =0x20026 ; ADP_Stopped_ApplicationExit
SWI 0x123456 ; ARM semihosting SWI
```

```
orr r1,r1,r2 ; base address 0x0 of Section;
```

Hao Sun, Advanced Microelectronic Systems Engineering, 2010 - 58 -

```
                                ; domain 3
str r1,[r0,#32];                SET it
ldr r2,=0x900000
ldr r1,=2_00000000000000000000100001110010;
orr r1,r1,r2                    ; base address TTBEentries section
                                ; User READONLY access
                                ; domain 3
str r1,[r0,#36];                SET it

; //=====
ldr r2,=0xa00000
ldr r1,=2_00000000000000000000110010010010;
orr r1,r1,r2                    ; base address UserHeap section
                                ; FULL access
                                ; domain 4
str r1,[r0,#40];                SET it
ldr r2,=0xb00000
ldr r1,=2_00000000000000000000110010010010;
orr r1,r1,r2                    ; base address UserHeap section
                                ; FULL access
                                ; domain 4
str r1,[r0,#44];                SET it
ldr r2,=0xc00000
ldr r1,=2_00000000000000000000110010010010;
orr r1,r1,r2                    ; base address UserHeap section
                                ; FULL access
                                ; domain 4
str r1,[r0,#48];                SET it
ldr r2,=0xd00000
ldr r1,=2_00000000000000000000110010010010;
orr r1,r1,r2                    ; base address UserHeap section
                                ; FULL access
                                ; domain 4
str r1,[r0,#52];                SET it

ldr r2,=0xe00000
ldr r1,=2_00000000000000000000110010010010;
orr r1,r1,r2                    ; base address UserHeap section
                                ; FULL access
                                ; domain 4
str r1,[r0,#56];                SET it
ldr r2,=0xf00000
ldr r1,=2_00000000000000000000110010010010;
orr r1,r1,r2                    ; base address UserHeap section
```

```

; FULL access
; domain 4
str r1,[r0,#60];          SET it

LDR r0, =TTBBase ; setup the table base register
MCR p15, 0, r0, c2, c0, 0 ; set table base address in cp15 r2

LDR r0, =2_00000000000101010100 ; set domain access to check by section entry
MCR p15, 0, r0, c3, c0, 0 ; commit r0 to co-processor15 register 3 (Domain access
control reg) ActivateMMU ; turn on the MMU!
MRC p15, 0, r0, c1, c0, 0 ; get the current control register
ORR r0, r0, #0x01 ; set the MMU flag
MCR p15, 0, r0, c1, c0, 0 ; commit the change!
NOP
NOP ; takes 2 cycles to activate MMU
ldmfd sp!,{r0-r12,lr}
mov pc,lr

```