Summary

Although the ARM processor has recently updated to multi-core, the legacy of a shared memory system indicates that there is currently no easy way to produce an efficient parallel implementation of algorithms, which will require to be investigated.

The aim of the project is to investigate the parallel efficiency of software implementation on conventional multi-core hardware with shared memory architecture. Two widely used software implementation shared memory programming model and message passing programming model are taken into comparison on Cortex-A9MPCore processor provided by ARM Company.

Various implementations are employed into this project. Since two programming models require to be compared in the project, a basic operating systems is established to support for these two programming model, which involve the cache coherency, virtual address and process management implementation. As for the implementation of two programming model, shared memory model adopts lock and unlock operation to maintain synchronous operations while message passing model utilize blocking communication to ensure the synchronization in communication.

Comparison results indicate that there is no significant performance gap between two programming models. Although shared memory model presents slightly better performance than message passing model, message passing model has potential to generate better performance when taking future processor and memory development into consideration.

- I establish a basic operating systems that acting like platform for shared memory and message passing implementation. Section 5.1
- I create basic send/ receive operation for message passing implementation and lock/unlock operation for shared memory implementation. Section 5.2
- I convert quick and merge sort into parallel program that can be operated on the two implementation above, and utilize it for comparison results. Section 6

Acknowledgements

I would like to express my deep gratitude to Dr Simon Hollis for his help in the project and for providing the software tool for the 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 would like to thank my friends who always willing to help me.

Table of contents

1. Motivation, aims and objectives

Recently, ARM processor has updated to multiple cores. However, the shared memory architecture indicates no approach exists to provide an efficient parallel implementation of algorithms, which are require to be investigated.

There are mainly three aims and objectives for this project

- Construct basic operating systems as a platform for both parallel programming models.

  As the operating systems for multi-core, the cache coherency protocol and process management should be taken into consideration. Also virtual address can be implemented to achieve better memory management.

- Create corresponding functions for shared memory model and message passing model.

  For message passing model, the basic send and receive function require to be established while for shared memory model, lock and unlock function can be used to prevent shared variable from inappropriately access.

- Obtain the comparison results by using parallel programming quick sort and merge sort.

  Two sorting algorithms will be parallel programmed and employed as evaluation tool to obtain comparison results.

2. Introduction to Parallel Hardware Architecture

This chapter is intended as an introduction to the parallel computer system architecture and memory architecture, which present the brief idea of parallel hardware architecture. Then the Coxtex-A9MPCore multicore processor, hardware platform for this project, has been studied.

2.1. Classification of Parallel System Architecture

Obviously, a variety of new computer architectures have been introduced for parallel processing in last sixty years, which expand the approaches to achieve better computation performance. However, this also increases the demands of the taxonomy to claim what kind of the parallel hardware architecture having already been established and the conjunction between them. Flynn's taxonomy, despite of the fact that it is one of the oldest classification of parallel computers, proves itself to be the most popular parallel computer architecture still widely used in these days.

In 1966, Michael Flynn classified systems based on two important elements, the number of instruction steams executed and the number of data streams manipulated simultaneously [1], which becomes an extremely useful tool to classify the category of the computer architectures for parallel processing. The simplest architecture in Flynn's taxonomy is the usual sequential computer, which identified as single-instruction single data architecture (SISD). On the contrary, it comes to the multiple-instruction multiple-data architecture (MIMD). Besides of these two, there are still the single-instruction multiple-data architecture (SIMD) and the multiple-instruction single-data architecture (MISD).

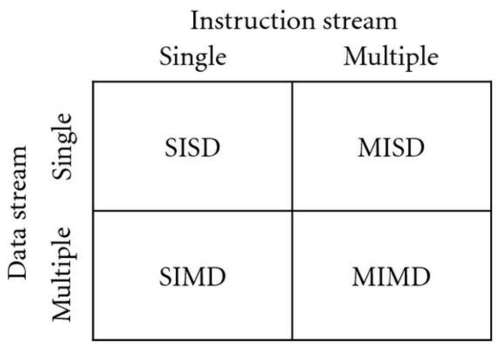


Fig 1 Flynn's taxonomy [1]

SISD architecture represents for the standard sequential computer. The single processor operates on single stream of data based on the single stream of instruction it received.

For instance, to obtain the result of the sum of N numbers, SISD architecture needs to access to the memory N times, also executes the addition instruction N-1 times based on the data it fetches from the memory. This indicates that SISD architecture doesn't contain any parallelism due to it only hold one processor.

MISD architecture represents the computer architectures that contain multiple processors, each processor with its own control unit for different instructions and shares the memory together. Parallelism on this architecture can be accomplished by manipulating different processor independently operating different instructions on the same stream of data simultaneously. It can be useful when dealing the computation where same input data requires to be handled with different operation.

For example, when encounter the problem that has to check whether X is a prime number. By calculating all possible division of X on different processor, the result probably can be achieved in one step. But taking other computation into consideration, the MISD architecture becomes quite difficult to handle with. Due to this fact, few actual examples of this architecture have ever existed [2].

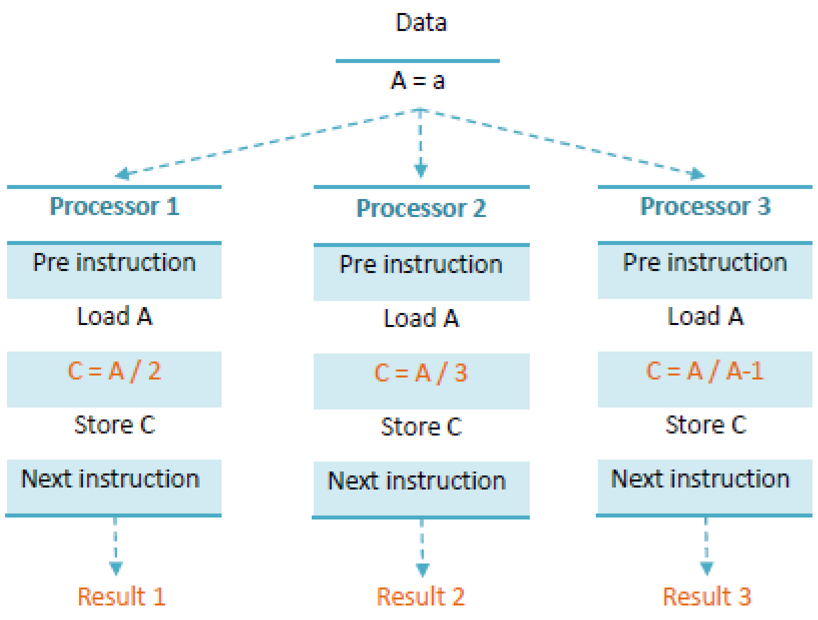


Fig 2 MISD Architecture

The SIMD architecture contains a single CPU as a central control unit with a large number of subordinate ALUs to manage the computation. It can be refer as N identical processors all controlled by a common control unit to operate the same program on different data streams. In order to achieve parallelism, these processors have to be operating synchronously on different data streams, which is more executable for most of the parallel computation.

SIMD architecture is quite suitable for dealing the problems with high degree of regularity, such as image processing [3]. However, the disadvantage of SIMD is that a program with too many conditional branches can cause many processors remain idle for long period of time. Lots of modern computers employ SIMD units, especially those with graphics control units (GPUs)

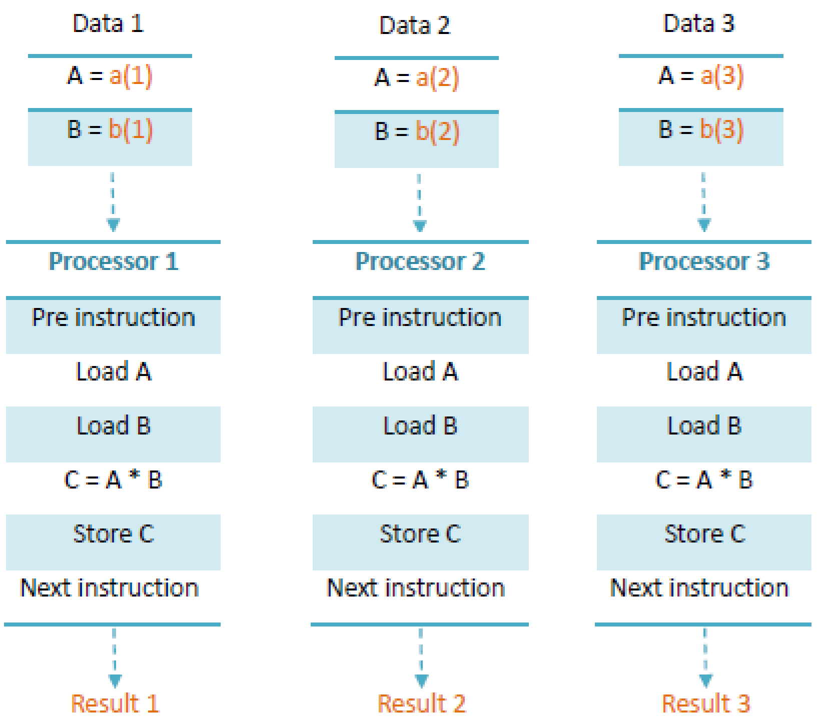


Fig 3 SIMD Architecture

MIMD architecture is affirmatively the most powerful architecture of the Flynn's taxonomy. Based on the fact that each processor holds its own control unit and local memory, it can operate different instruction streams on different data streams.

So with MIMD architecture, computer has the privilege to execute synchronously or asynchronously, deterministic or non-deterministic. It should also be indicated that MIMD computer can be either shared memory architecture or distributed memory architecture, which can be classified based on how MIMD processors access memory. Most current supercomputers fall into MIMD category, such as Intel IA32, IBM POWER5 [3].

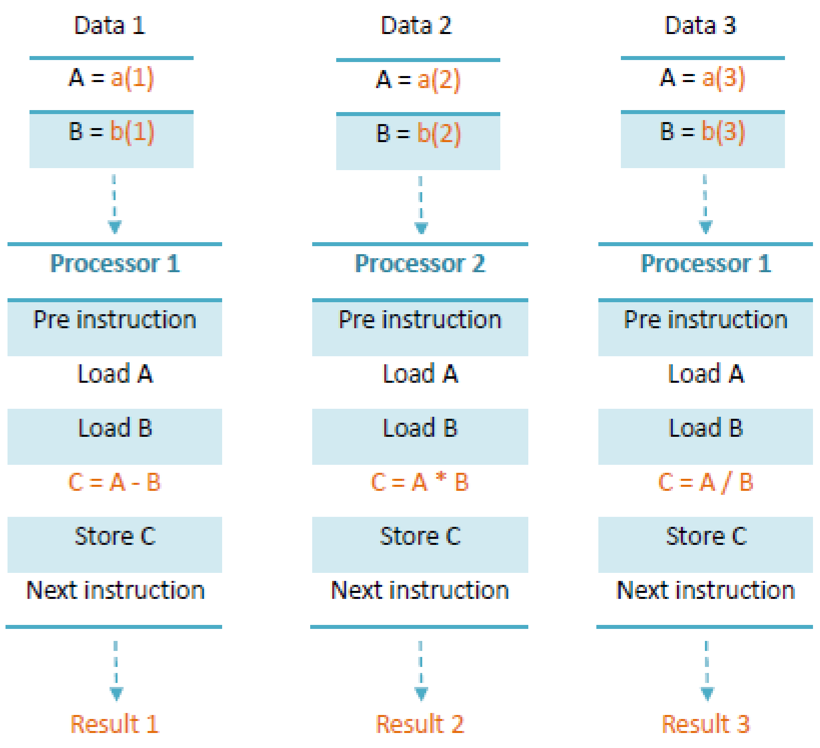


Fig 4 MIMD Architecture

2.2. Parallel Computer Memory Architecture

There are mainly two kinds of parallel computer memory architecture nowadays, shared memory architecture versus distributed memory architecture.

2.2.1. Shared Memory Architecture

Shared memory architecture refers to the structure which holds a block of random access memory that can be accessed by multiple processors. It is capable of providing a global address space for all the processors in multiple-processor computer system, which each processor can operate independently but share the same memory resources. Any changes in shared memory location updated by one processor are visible to all the other processors [4]. As a matter of the fact, communication between processors can be accomplished by updating and fetching data from the same location in global address space.

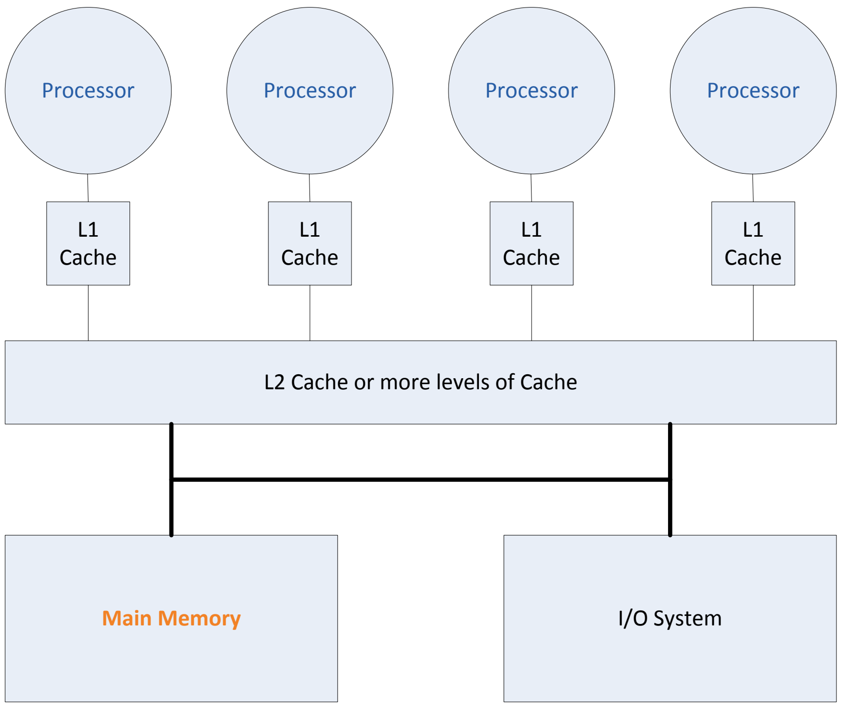


Fig 5 Shared Memory Architecture

Based on memory access time, the shared memory architecture can be classified into two categories, uniform memory access (UMA) and non- uniform memory access (NUMA).

In UMA architecture, the time access to a memory location is independent of which processor requires the data from the memory or which memory contains the data demanded to be transformed. Each processor is identical and has equal access privilege and time. In contrast, the time access to the memory depends on the relationship between processors and memory location. Memory access across the link usually consumes more time.

As shared memory architecture provides a global address space, it is relatively easy programmer to design a parallel program, and data transformed between processors can be both fast and uniform. However, there are mainly two issues in shared memory architecture, lack of scalability and cache coherence overheads.

Scalability between processors and memory becomes a bottleneck for shared memory architecture, as adding the number of processors leads to geometrically traffic increased on the processor to memory path. So multiple-processor computer systems with shared memory architecture usually have ten or fewer processors.

Cache coherence overheads can be the other primary disadvantage of shared memory architecture. As nowadays each processor has its own cache which can be assumed as fast, small block of the memory located between processor core and main memory. And it temporarily holds a subset of data and instructions in the main memory to speed up the access time of the memory. Whenever a processor access a shared variable through its own cache, it is difficult for this processor to acknowledge whether the value stored in variable is current.

Cache coherency protocol is essential to solve such conflicts and maintain the data stored in cache consistency [3]. Such cache coherency protocols guarantee the correct communication between multiple processors under shared memory architecture. But they can sometimes become overheads, which affect the multiple-processor computer system to achieve high performance.

Besides two primary disadvantages which degrade the performance of the system, the memory design tend to be more complicated and the programmer also need to responsible for the synchronization constructs to prevent inappropriately access to the global memory. Nowadays, with ever increasing number of processors, it becomes more expensive and difficult to design multiple processor computers with shared memory architecture.

2.2.2. Distributed Memory Architecture

Distributed memory architecture refers to the structure that each processor has its own private memory, and communication network is allocated to connect inter-processor memory [4]. Because each processor can operate independently on its own private memory, there is no concept of global address space across all processors and cache coherency since modifications on its private memory have no influence on the memory of other processors.

Due to the fact that each processor holds its own private memory in distributed memory architecture, the memory is capable of scalable with the number of the processors that making systems with high computing power possible. And since each processor can access its own private memory rapidly, there is no requirement for the existence of the cache coherency protocol, which reduce the overhead in performance of the multiple processor computers to maintain cache coherency.

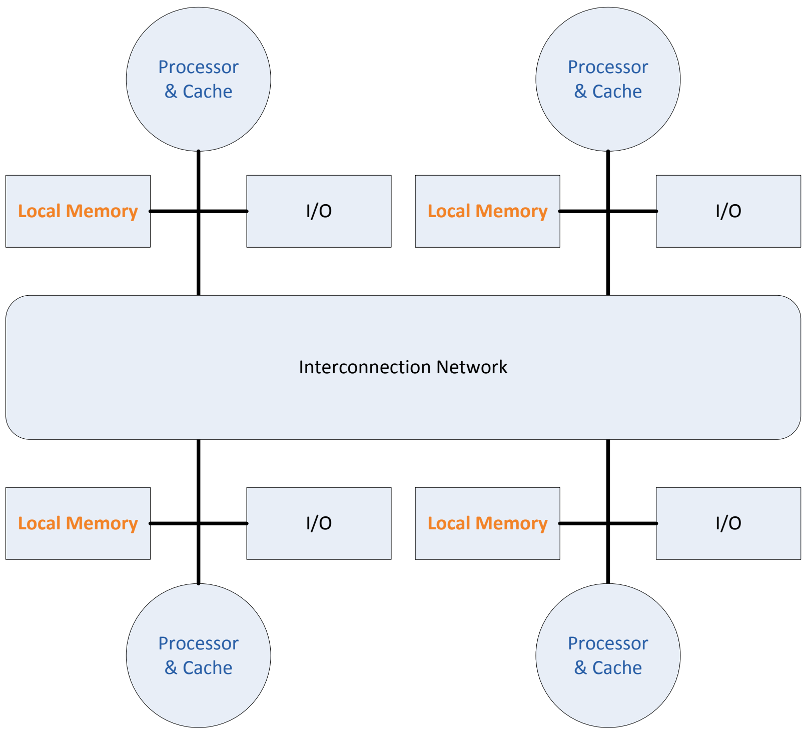


Fig 6 Distributed Memory Architecture

However, distributed memory architecture also has two main drawbacks, communication program difficulty and memory organization difficulty. Communication between processors in distributed memory architecture is achieved by message passing through interconnection network, which involve send and receive operations. Data distribution and communication management should all be taking care of in these operations. This creates too many details for programmer to handle and increase the difficulty for the programming. Also the memory organization becomes difficult for the mapping of the existing data structure [3].

Therefore, by taking two memory architecture into consideration, shared memory architecture has complex hardware problems but easier software problems while distribute memory architecture has less hardware problems but complicated software problems.

2.2.3. Hybrid Distributed-Shared Memory Architecture

As both memory architectures have its own advantages and disadvantages, a relatively new design, Hybird distributed-shared memory architecture, has been carried out to combine the advantages of both memory architectures and reduce the effect caused by disadvantages. Recent studies indicate that this architecture seems to have potential to increase the performance of parallel computing for the foreseeable future [3].

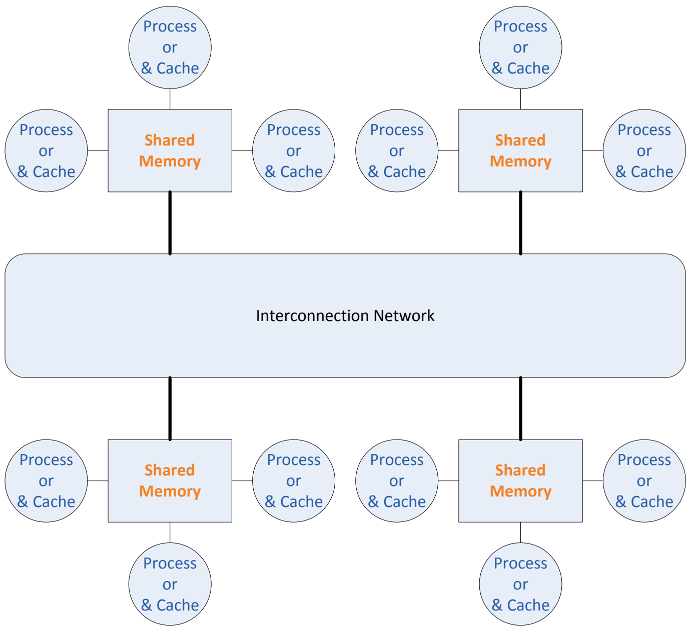


Fig 7 Hybird Distributed-Shared Memory Architecture

2.3. Introduction to Coxtex-A9MPCore Multicore Processor

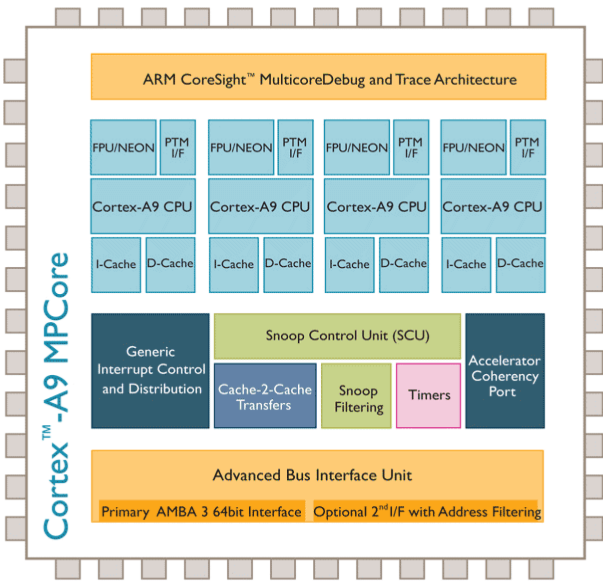

ARM CoreSight™ MulticoreDebug and Trace Architecture
FPU/NEON
PTM I/F
Cortex-A9 CPU
I-Cache
D-Cache
Generic Interrupt Control and Distribution
Snoop Control Unit (SCU)
Cache-2-Cache Transfers
Snoop Filtering
Timers
Accelerator Coherency Port
Advanced Bus Interface Unit
Primary AMBA 3 64bit Interface
Optional 2nd I/F with Address Filtering
Cortex™-A9 MPCore

Fig 8 Cortex-A9 MPCore Multicore Processor Configuration [5]

The investigation of the software implementation efficiency is studied on Cortex-A9MPCore multicore processor manufactured by ARM Company. In order to obtain brief idea about the hardware platform involved in the project, the hardware architecture of Coxtex-A9MPCore will be discussed.

The Coxtex-A9MPcore multicore processor provides a hardware architecture supporting 1 to 4 processors in an integrated cache coherent manner [5]. Each processor can be independently configured with its own cache, floating-point unit (FPU) and NEON media processing Engine (MPE). In addition, accelerator coherence port (ACP) permits other non-cached system-mastering peripherals to be cache coherent with processor L1 cache.

Snoop Control Unit (SCU) acts as central communication control unit in ARM multicore hardware. And it is responsible for managing interconnection, communication and cache coherence, which are all significant in multiple processors technology.

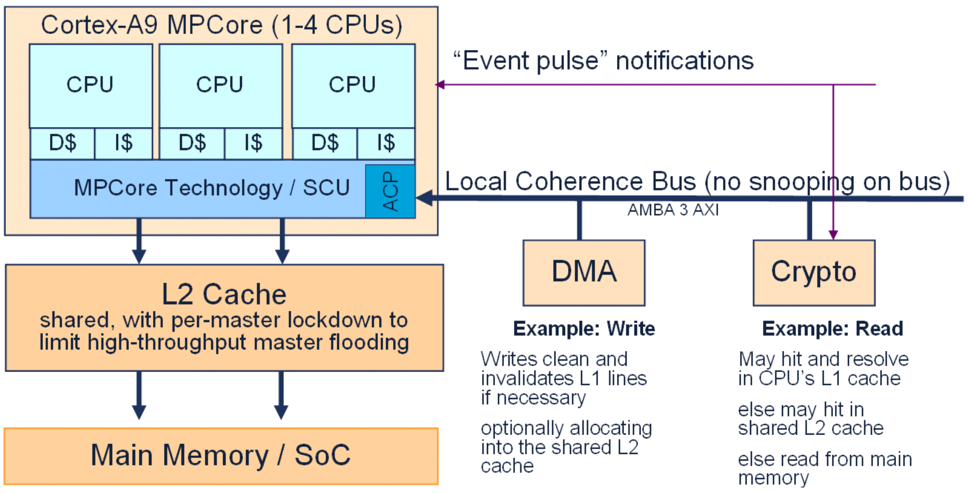

Cortex-A9 MPCore (1-4 CPUs)
CPU
CPU
CPU
D$
I$
D$
I$
D$
I$
MPCore Technology / SCU
ACP
"Event pulse" notifications
Local Coherence Bus (no snooping on bus)
AMBA 3 AXI
L2 Cache
shared, with per-master lockdown to limit high-throughput master flooding
Main Memory / SoC
DMA
Crypto
Example: Write
Writes clean and invalidates L1 lines if necessary
optionally allocating into the shared L2 cache
Example: Read
May hit and resolve in CPU's L1 cache
else may hit in shared L2 cache
else read from main memory

Fig 9 SCU Operation Procedure [5]

Any operation on coherent memory region will interact with snoop control unit first [5]. For the read operation, the snoop control unit tests whether the required information is already stored in processor L1 cache. If the required information can be located, it returns the information needed. Otherwise, there is still an opportunity to hit in L2 cache, before it finally forward to the main memory. For the write operation, the snoop control unit will enforce coherence before information is written to the memory system.

Generally in Cotex-A9MPCore multicore processor, four processors shared the global memory together with two levels of caches. Each processor holds its own L1 cache, and L2 cache is shared among processors. Snoop control unit are employed to maintain cache coherency in communication.

3. Introduction to Parallel Programming

In this chapter, the challenges in parallel programming are summarized at first, and then several widely used parallel programming models are discussed in order to present the brief idea about parallel programming.

3.1. Challenges in Parallel Programming

In order to increase performance and efficiency through parallelism, there are five challenges need to taking into consideration.

A. Independent operation in application

Finding independent operation in an application can be the first challenge for parallel programming. As independent operation can be operated without concern about other operation resources or results, the parallelism efficiency can be highly improved by programming several independent operation execute concurrently.

Some applications hold large amounts of data parallelism, which can be divided into several sets of data element and execute independently. While other applications have great number of task parallelism in which different operation can be performed in parallel. But in both data and task parallelism, there are usually still contain a sequence of dependent operation which limited the parallel performance.

B. Communication between operations

Second challenge, communication between operations, occurs when the computations of the operation are not entirely independent and needs to communicate with each other. Even some application can fully divide into different independent operations, the application still need the parallel operations to communicate with each other to find the solution.

For instance, assuming there is an array need to be sorted. First the program needs to divide the array into different subset and distribute each subset to different processor for further operation. Then after each processor finish its own computation, one processor requires to gather all subsets in every processor to form the result. This is when the communication taking place.

The parallel programming can achieve communication by reading and writing to shared memory space or by sending and receiving message between parallel operations. However, both approach need to confront the communication overhead problems.

C. Keep locality among operation

Keep locality among operation is the third challenge in parallel programming. By placing two operations that access the same data near each other in space or in time, locality can be implemented to reduce the costs caused by communication [6].

Placing the operation nearby in space can reduce the distance data have to transfer. While placing the operation nearby in time can reduce the amount of live data, which could be stored in fast small memories, like cache. Locality not only reduces the demand for communication between processors, but also tremendously decreases the costs of another communication, data movement between processors and memory.

D. Operation synchronization

Synchronization, as the fourth challenge, is needed to provide cooperation mechanism between parallel operations. As some operation execute dependent on other operation results. Synchronization is used as a safety protocol to guarantee the parallel operations execute in the correct order. There are often conflicts between performance improvement from parallelism and computation correctness by synchronization [6].

Taking some synchronization operation for example, lock and unlock operation are provided to ensure only one processor can modify the shared variable each time, and the variable can only be unlocked by its locker. Barrier synchronization forces all the parallel operations to wait until all of them reach the barrier. Because synchronization essentially reduce the performance gain increased by parallelism. It is important to make sure there is no unnecessary synchronization.

E. Balance the load of operation

Finally, load balancing can be the last challenge for parallel programming. Load balancing, distributing operations evenly among processors, can maximize the parallelism ability of system. On the contrary, if the load becomes unbalanced since some processors consume more time to fulfill their work. Other processors have to remain idle when program execute ends.

In fact, load balancing difficulty is determined by the characteristics of the application [6]. It can be either quite naïve if all the operations have the same cost, or extremely difficult if the cost of operations remain invisible until they are executed.

3.2. Introduction to Parallel Programming Model

Parallel programming model can be considered as the parallel software implementation, which exists as an abstraction above the hardware and the memory architecture. There are several common used parallel programming models.

- Shared Memory Programming Model
- Message Passing Programming Model
- Thread Programming Model
- Data Parallel Programming Model

It may seem illogical that these models are not specific to a particular type of system or memory architecture. However, in fact all these models can theoretically be implemented on any kind of memory architecture. For instance, Kendall Square Research ALLCACHE approach implements shared memory model on distributed memory machine with virtual shared memory approach, while SGI Origin 2000 applies distributed memory model on shared memory machine using message passing [3].

So it seems there are no limits in choosing the parallel programming model. However based on hardware resource available and practical problem complexity, some programming model implementations will have better performance than the other.

3.2.1. Shared Memory Programming Model

The general idea of shared memory model is that all the processors have the equal privilege to access all memory location. But as mentioned before, shared memory programming model can also be emulated on distributed memory architecture via virtual shared memory approach. It seems more precise to define shared memory programming model has the mechanism of sharing a global address space.

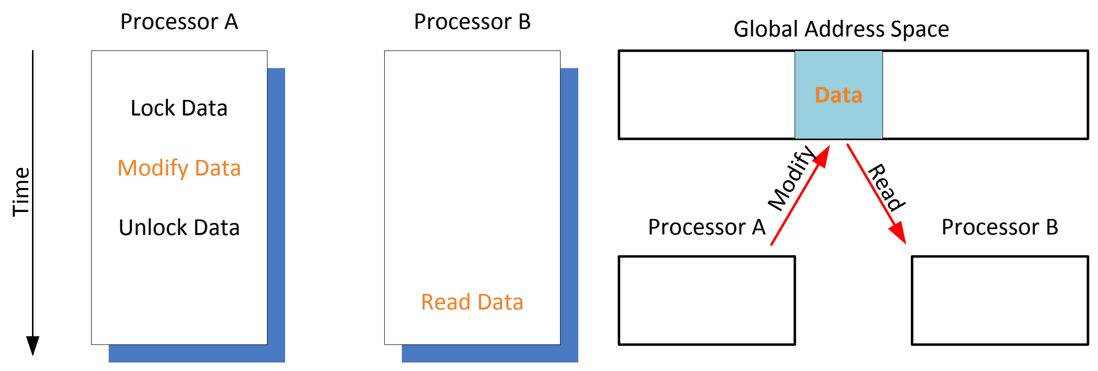


Fig 10 Basic Shared Memory Implementation

There are three basic principles to manage communication among processors in shared memory programming model [4].

- Shared variables should be able to be accessed by all the processors
- Offer an approach for processors operating synchronously.
- Prevent the processors from inappropriately accessing shared resources

In order to illustrate the idea of above principle, a simple shared memory addition program has been present. Supposed that each processor computes a private integer private_num, which can only be accessible by its own processor. The program needs to calculate the sum of private_num in each processor and print out the result.

According to the first principle, a shared variable sum, can be accessed by all processors, should be allocated in shared memory model. In this example, it will be simply defined with a prefix called shared, while in reality, this shared variable needs to be assigned to a specific global address space. The program will be simply the same as the sequential program at first.

But one significant issue can be drawn from the above graph is that the result of the sum is not the value expected. Actually, depending on the time each processor executes the addition of the private_num to the sum, the sum can be undetermined value 1, 2 or 3. This is due to the fact that each processor operates the program on its own without synchronization.

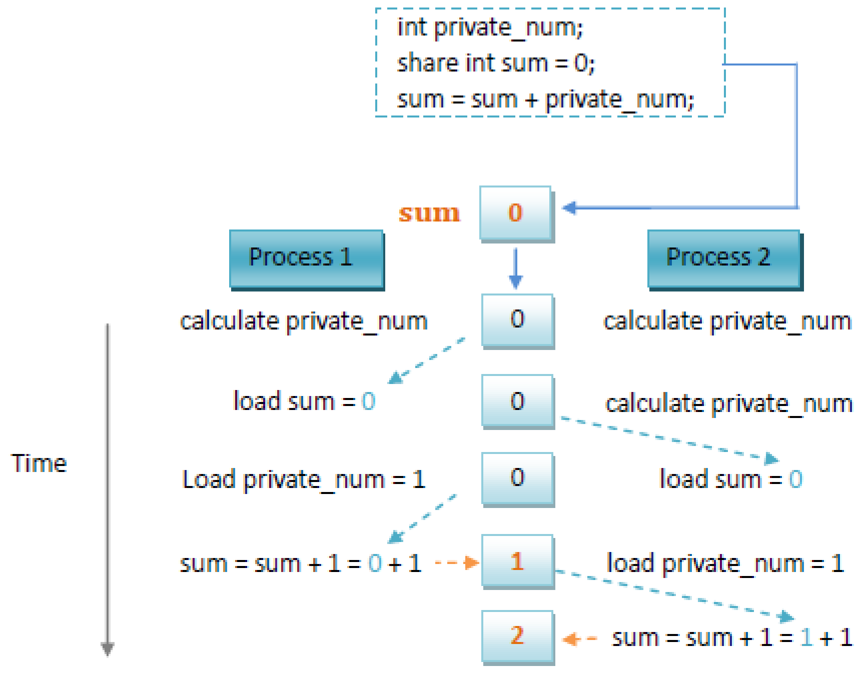


Fig 11 Shared Memory Addition Program [4]

Providing an approach for processors operating synchronously, as second principle in shared memory programming model, guarantee the processors reading and writing shared variables in right order. In this example, the code sum = sum + private_num should only be executed by one processor at a time to obtain the correct result. Code section like that which can only be accessed by one processor each time is defined as critical section.

Mutual exclusion has to be arranged to obtain the correct runtime order. And one of the simplest approaches to achieve mutual exclusion is called binary semaphore [4]. The fundamental idea of binary semaphore is creating a shared variable flag s, which indicates whether the critical section is free. For instance, if the value of s is 1, then the critical section can be accessed by processors. If the value is 0, the region is closed for access.

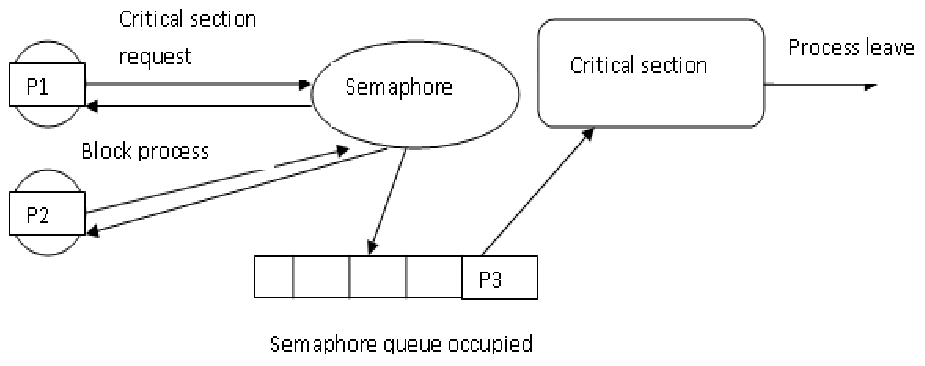


Fig 12 Semaphore Mutual Exclusion [6]

The problem still remains as the lack of protection of shared resource. Taking shared variable flags for example, when one processor is fetching s = 1 into its own register to test, the other processor may be storing the s = 0. In order to prevent the processors from inappropriately accessing the shared resources, lock and unlock mechanism will be required. Once a variable is locked, only the processor that locked it can modify the value in it.

The final issue widely occurs in shared memory model is that how could the processor in charge of the printing knows the time when the other processors complete their computation without communication. A high level operation called barrier [4] has been carried out to fix this problem. The barrier, acting like a function, will not return the value to the processors called it until every other processor has called it.

The advantage of the shared memory model is the concept of the data ownership is lacking. So there is no necessary to specify the communication of data between processors, which obvious simplified the program development work. However, when taking performance into consideration, shared memory programming model have difficulty in managing data locality, which will reduce parallel computing efficiency with processors number increased.

3.2.2. Message Passing Programming Model

Message passing programming model is the most commonly used programming model for distributed memory systems. But it can also be implemented on shared memory systems such as SGI Origin 2000. From a programming perspective, message passing model is to achieve parallelism based on existing message passing library consisted of communication functions and subroutines.

Although a great diversity of message passing libraries become available since 1980s, it makes programmer difficult to develop portable applications as each library has different implementation. Until 1994, Message Passing Interface (MPI) is released, which establish a standard interface for message passing implementations, making it possible for developer of parallel software writing both portable and efficient parallel program [3].

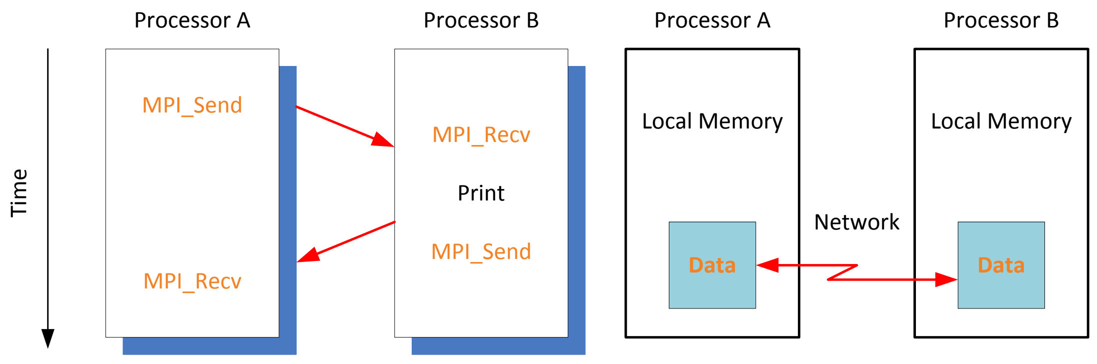


Fig 13 Basic Message Passing Implementation

For message passing model, there are also two primitives

- The processors communicate with each other through sending and receiving message.
- Data transfer usually requires cooperative operation being performed by every related processor.

In order to present some brief idea about how message being passed between processors, MPI function MPI_Send and MPI_Recv are presented as illustration.

When pass message between processors, the source and destination is provided to guarantee the message is transferred correctly. Also the message address and the size of the message must be provided to determine the location and the length of the message. This is the reason why buffer address, count and datatype are packed with message in communication. Besides, the tag is implemented to indicate what the operations need to performed on the message received. And the communicator is allocated to state the communication context, which is the collection of processes that can send message to each other [4].

After introducing what should be packed into message, communication mechanisms are taking into consideration. There are three basic communication mechanisms, buffered communication, blocking communication and non-blocking communication.

Buffered communication is applied to improve parallel program performance. Suppose that every send operation perfectly synchronized with receive operation, the performance of the program will be remarkable. But in reality this is rarely the case. When the send operation is executed before receive operation, the processor in charge of the send operation usually has to remain idle until the receiving processor is ready to receive message, which obvious reduce the efficiency of parallel program.

Alternatively, buffered communication can be implied to solve this problem. It provides a system-controlled memory, utilized as transfer station, to store the message. This gives the send receive operation privilege to be asynchronous by free the sending processor from waiting. However it is not always the case since the copying between the buffer and the processor memory location can be longer than the receive operation arrive, which trades off idling overhead for buffer copying overhead.

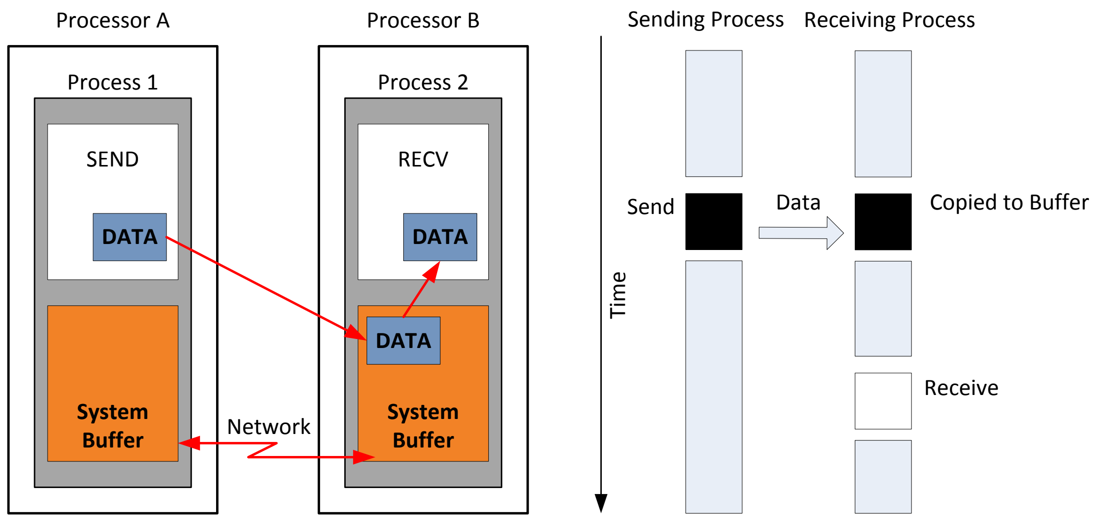


Fig 14 Buffered communication

Blocking communication guarantees the send and the receive operations only return after the data is semantically safe. The buffered blocking send only return after the data is safely located in system buffer and the processors memory can be reuse without affecting the data for receive processor, while the blocking receive only return after the data is arrived and ready to use for the computation.

In order to achieve better parallel efficiency, another communication mechanism, non-blocking communication is provided. Non-blocking send and receive operation return immediately without waiting communication completed. In MPI library, the receive function MPI_Irecv, instead of MPI_recv, creates one more parameter request [4]. After the system initialize the request argument, the processor can return for other useful work and will check back later to see if the message has arrived.

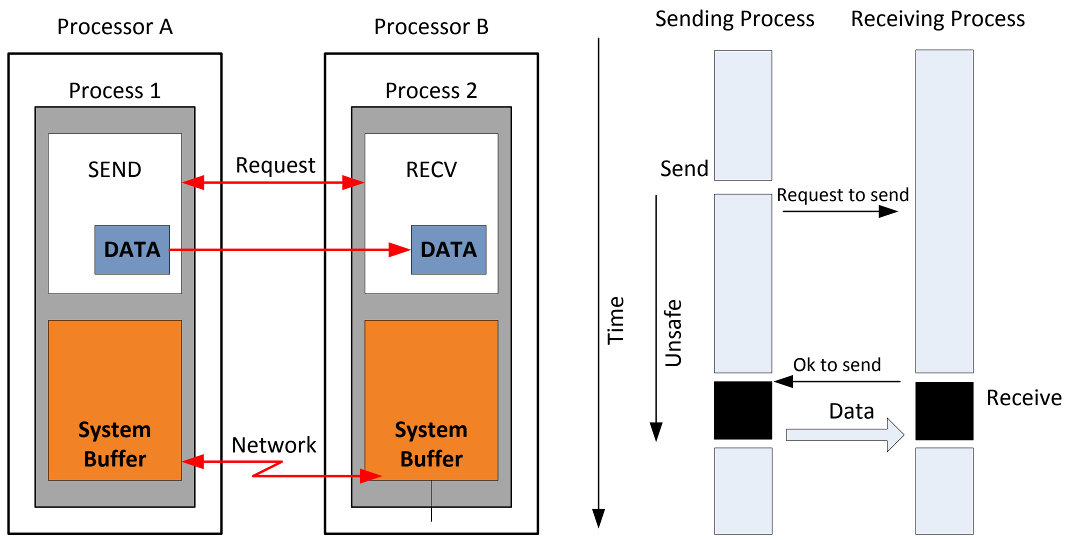


Fig 15 Non-Blocking communication

Non-blocking communication achieve dramatic performance gain by overlapping communication with computation. But one major disadvantage of non-blocking communication is that it will be unsafe to modify the data in processor memory until the request non-blocking operation is actually performed.

3.2.3. Other Parallel Programming Model

Besides shared memory programming model and message passing programming model, there are still several widely used programming model, for instance, thread model and data parallel model.

Thread model is one type of shared memory programming. Unlike the processes don't share resource, multiple thread can exist in the same process and share the memory together, which can be scheduled by operating system and execute concurrently. One significant advantage of thread model is that if one thread has a lot of cache misses, other thread can utilize it. This leads to better cache usage and efficient performance [7]. But, thread scheduling and interface can be serious problems for thread programming model.

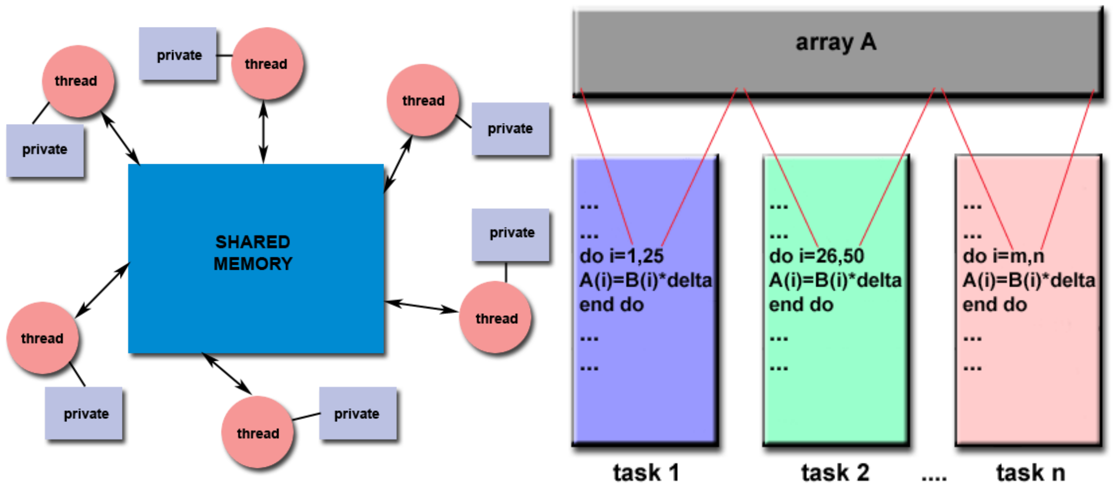


Fig 16 Thread Model and Data Parallel Model [7]

Data parallel model focus parallel operation on data set, which is usually, organized in common data structures, such as array or cube. When a set of processors operate on a data structure, each processor can be distributed with different partition of the structure and execute independently on it. In shared memory architecture, each processor can access the data through global address space; while in distributed memory architecture, each processor operates on the subset of the data structure after the distribution step.

There are still large amounts of parallel programming models available nowadays. Programmer should make the decision depends on the hardware architecture available, practical application complexity and parallel performance analysis.

4. Comparison between MP and SM Model

This chapter focus on two most widely used parallel programming model Shared Memory model and Message Passing Model. Both will be taking into comparison on different hardware architecture, especially on shared memory architecture, which intend to reveal the strongpoint and limits of both parallel programming models.

4.1. Previous Work on the Comparison

The common idea of parallel programming model is that shared memory model should be implemented on the shared memory architecture, while the message passing model should be implemented on the distributed memory architecture. However, theoretically both models can be implemented on either of memory architecture. So which model should be implemented will be determined by parallelism performance.

4.1.1. Implementation on Distributed Memory Architecture

As mentioned before, shared memory model can be implemented on distributed memory architecture by defining a global address space. Global Arrays, designed by Jaroslaw and Robert, successfully establish a shared memory programming model on distributed memory machine [8]. The global array provides a portable interface for every processor in MIMD parallel program. So each processor can access the logical blocks of physically distributed matrices asynchronously [8], which is quite similar to shared memory model.

But taking parallelism performance into consideration, message passing model seems more suitable for distributed memory architecture, which reduces the demands for shared memory model to implement on distributed memory machines.

Since the project purpose is to investigate the parallel efficiency of software implementation on conventional multi-core hardware, which is supposed to have shared memory architecture. More background studies about implementation on shared memory architecture will be introduced below.

4.1.2. Implementation on Shared Memory Architecture

Although message passing model is initially designed for distributed memory architecture, it can also be implemented on shared memory architecture. So the parallelism efficiency will dominate the choice of programming model.

A. C.Lin and L.Snyder Work

When the implementation of message passing on shared memory architecture was first investigated by C.Lin and L.Snyder [9], the result is unexpected. Some case shows that there are nearly no difference in performance between two models, while other cases indicates the surprisingly fact that message passing programming model tend to be more efficient than shared memory programming model. Although the result may show potential ability of message passing implemented on shared memory architecture, it is not convincing enough since the simplicity of the program and limits of machine.

B. Ton A.Ngo and Lawrence Work

Then the implementation was further investigated by Ton A.Ngo and Lawrence [10]. They simulate more complicated algorithms optimized LU on three different kind of parallel machines, the Sequent Symmetry, the Cedar and the Butterfly.

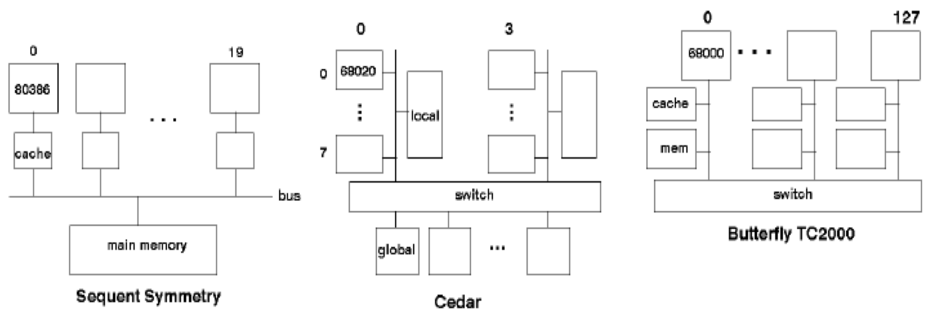


Fig 17 Symmetry Cedar and Butterfly Hardware Architecture [10]

The Sequent Symmetry is a small scaled bus-based machine. The bus is capable of holding 20 nodes since the processors speed is quite low. Cache coherency is managed by snooping using Illinois protocol [10].

Cedar has cluster architecture. Each bus-based cluster contains 8 processors sharing a local memory and all the clusters are connected to the global memory through a switch [10]. Therefore, unlike Sequent Symmetry have two levels of memory hierarchy.

For the Butterfly machine, each processor has its own cache and local memory. And each local memory of nodes is connected to the others through interconnection network. Therefore, the global memory consists of the aggregate of all local memories. Since there is no hardware coherency mechanism, the machine provides various cache invalidation functions to support software caching [10].

Although both parallel programming models is established based on the same LU algorithm, the approach they dealing with the data are different.

In shared memory model, the data reside in the global memory. And all processor can access to the global memory and update any segment of matrix [10]. Standard lock and barrier are implemented for synchronization. While in message passing model, data are placed in the level of the memory closest to the processor. With buffer located in the global memory, send and receive operation are emulated with block copy. Synchronization in message passing model is implicit since blocking sends and receives are used [11].

Consider about the complexity of the LU algorithms and different hardware architecture machines involved, the result should be more convincing. Figure 18 indicates the performance of LU decomposition on three machines for three problem sizes.

As the performance graph shows, on the machine Sequent Symmetry, the shared memory implementation has slightly better performance than message passing implementation. However, for the machine Cedar, the message passing implementation begins to provide better performance. And when it comes to Butterfly, there is large performance gap between two models implementation due to the large gap in access latency [11].

The distinction is due to the reason that parallel machine Sequent is lack of memory hierarchy while Butterfly has deep memory hierarchy. Performance result indicates the fact that message passing favors to be implemented on shared memory machines with private processors cache [11], where it advantage data locality can be fully exploited.

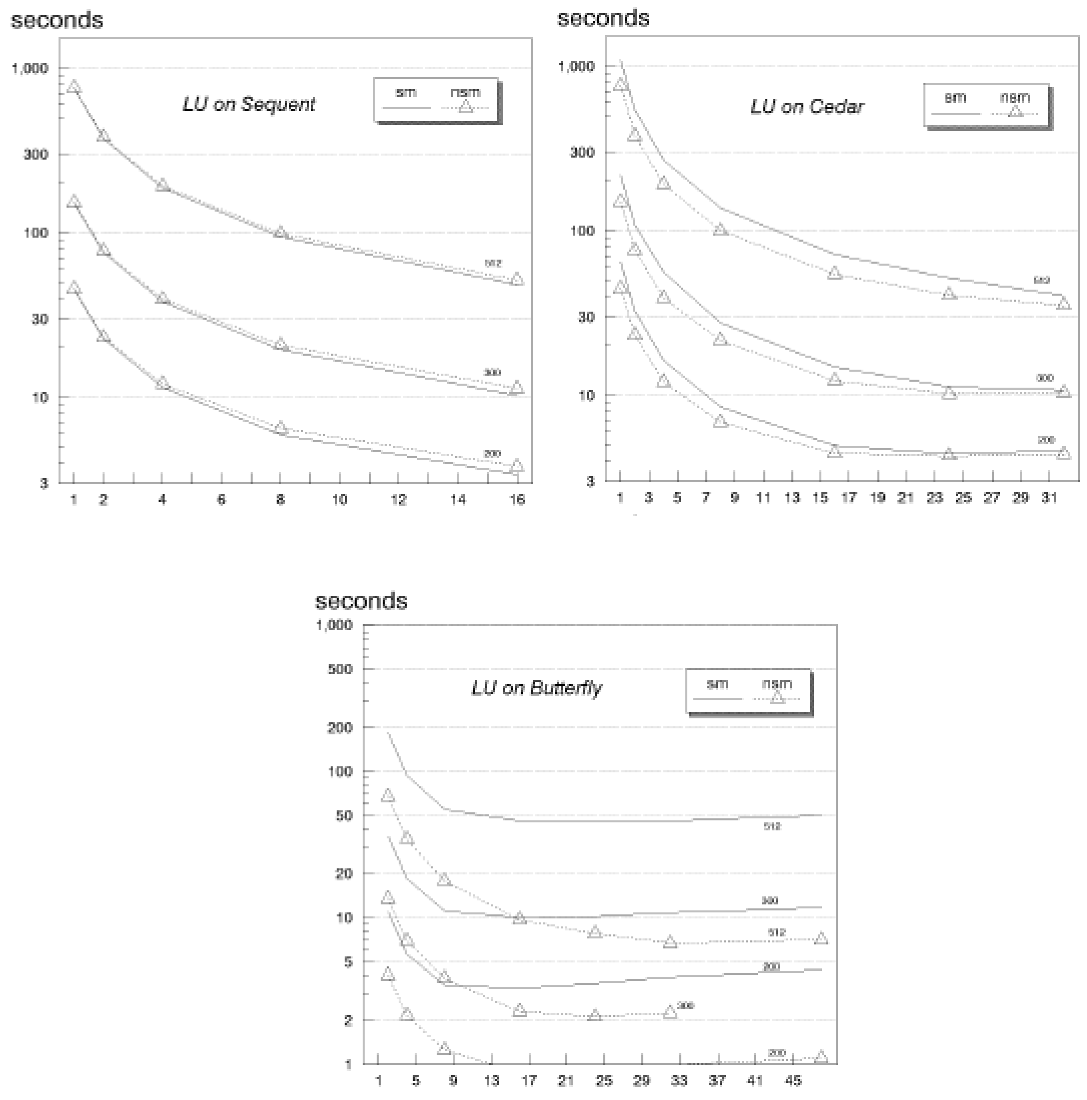


Fig 18 SM and MP Performance Comparison [11]

C. LeBlanc and Markatos Work

Since the result is quite unexpected, more specific and advanced studies have been placed on this area. Although Ton A.Ngo and Lawrence' result is concluded based on complicated algorithm and wide range of shared memory architecture implementation, LeBlanc and Markatos indicate that one significant fact have been neglected, the load imbalance [12].

Load imbalance, one of the five challenges in parallel programming, is mainly caused by two reasons in parallel program. Assigning computation to the process or thread unequally can be one reason, while assigning processes to the physical processor unevenly can be the other. As it is quite reasonable to assume that inherent load imbalance can potentially exist in applications, the ability to manage the load imbalance in applications should be taken into consideration when comparing two parallel programming models.

In shared memory model, the effect caused uneven assignment can be minimized by using lightweight thread [12]. In order to avoid uneven assignment of the threads to processors, a central ready queue is established to distribute the load among processors []. Every idle processor can remove the thread from the central ready queue for execution, which guarantees no processors remain idle until program is finished.

While in message passing model, the data is static divided in processes and the processes are static distributed to the processors, which can lead to load imbalance. Although load balancing can be achieved by applying process migration, but the cost can be quite expensive.

LeBlanc and Markatos first experiment application with little inherent load imbalance on six different shared memory machines. It well proved the result studied by Ton A.Ngo and Lawrence since message passing implementation performs better in nearly all the cases. Then the applications with extreme load imbalance are introduced to the experiment.

As it can be seen in figures, the shared memory implementation has slightly better performance than message passing implementation on Butterfly, and much better performance than message passing implementation on Symmetry. As both machines have enormous bandwidth and relatively slow processor speed, the advantages of shared memory model dominate on these machines [12].

However, message passing implementation performs better than shared memory implementation on Iris machine even significant load imbalance exists. Since Iris holds less bandwidth and faster processors compared to Symmetry, the communication can be relatively expensive. In this case, locality, message passing implementation primary advantage, begins to dominate the parallel performance.

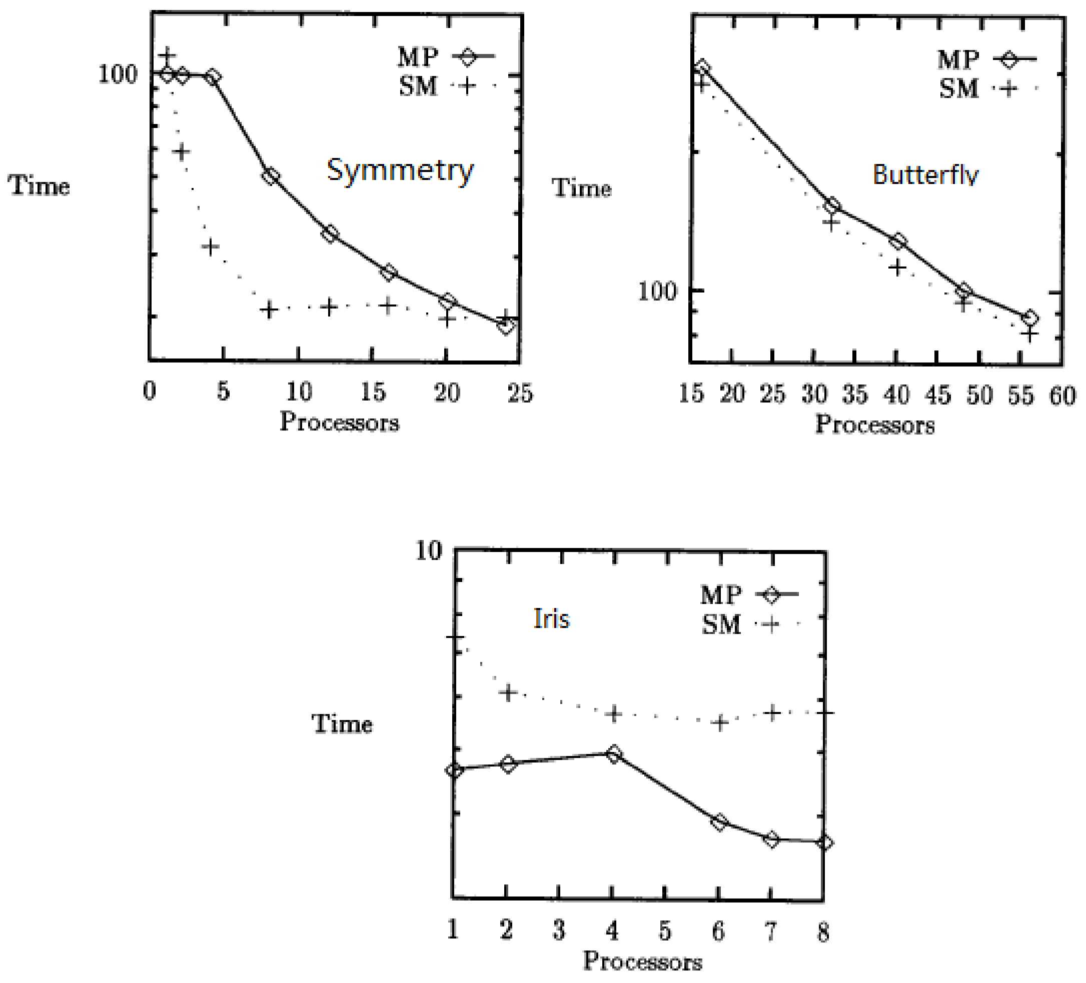


Fig 19 SM and MP Performance Comparison with Extreme Load Imbalance [12]

As message passing model is more preferable with no load imbalance exists and shared memory model performs better under extreme load imbalance conditions, both models seem to have its own advantages and limits. In order to examine how the degree of load imbalance affects the choice between two models, LeBlanc and Markatos repeatedly execute the transitive closure application by using different inputs to vary the amount of load imbalance on two machines, Symmetry and Iris [12].

Since the approach of modifying load imbalance is implemented by using different transitive closure application inputs. LeBlanc and Markatos use 1000 nodes as input, where first N nodes forms a clique, and the other remaining nodes have no connection [12]. The load imbalance can be adjusted by vary N from 100 to 1000. When the proportion of nodes in clique increases, the load that distributed among processors increase, which decline the degree of the load imbalance. On the opposite, if the proportion of nodes drops to zero, the extent of load imbalance is maximized.

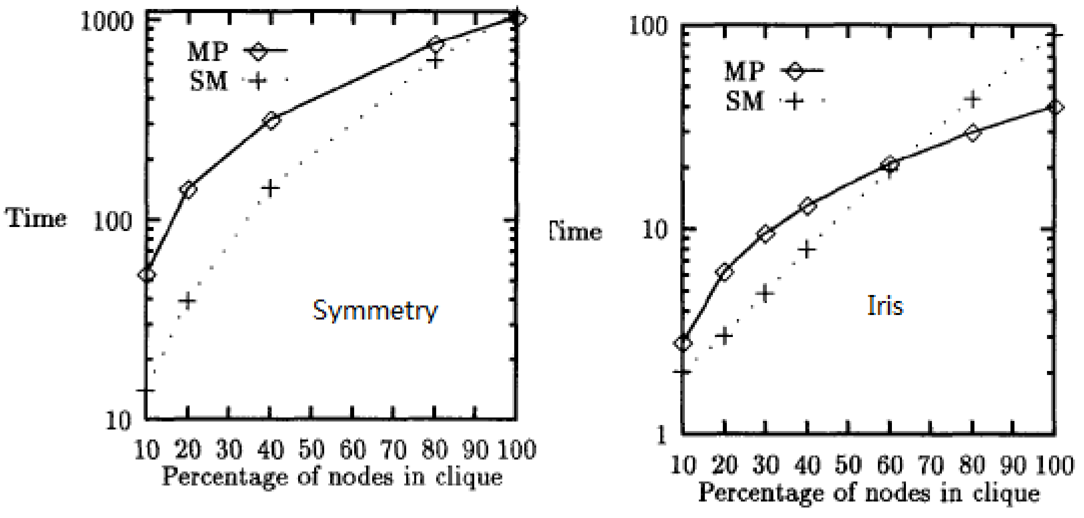


Fig 20 SM and MP Performance Comparison with Precise Load Imbalance [12]

The figure 20 indicates that for the Symmetry, the shared memory implementation has better performance over a wide range of load imbalance. While the message passing implementation only performs slightly better until there is no load imbalance exists. Nevertheless, the exact opposite occurs on Iris Machine. The message passing implementation holds better performance across wide range of load imbalance. It can be twice as fast as shared memory when the load is distributed evenly. Even under extreme load imbalance condition, the message passing implementation is only about 25% worse than shared memory implementation.

This distinction is caused by the reason that the cost of communication is much higher on Iris than it on Symmetry. So to exploit data locality becomes more important than to maintain load balance.

D. Further research

Since strengths and limits of both models have been systematic investigated, further researches on message passing and shared memory implementation can be separated into two group. One group of researchers attempts to integrate the advantages of two models to construct a new model with better parallelism performance [13] [14]. While the other group spends the effort on optimizing parallel implementation based on specific hardware architecture [15].

4.2. Data Locality versus Load Balancing

The comparison between message passing models and shared memory models can be explained as the comparison of two strengths, keeping data locality for message passing implementation and maintaining load balancing for shared memory implementation.

Message passing model benefits from the data locality because it can more completely exploit the faster local memory. However, the benefit varies with the machine architecture and depends mainly on the effective gap between the global and local memory access time [11]. As processors load and store data to local memory is normally much faster than accessing global memory. For message passing implementation on shared memory architecture, the way it transmitting packet message thought global memory but operating data on local memory usually obtain better performance in most of the application.

Shared memory model performs better on the application with great extend of load imbalance. By using central ready queue as dynamic load balancing policies, the inherent load imbalance in application can be rearranged and distributed evenly to processors. But in message passing model, the data is static divided in processes and the processes are static distributed to the processors. This leads to inherent load imbalance of applications continuing to exist in the program, which cause some of the processors are idle most of time. Generally shared memory model adopts cheap thread that may execute on any processors and access data in global memory with all the resources, while message passing model employs heavyweight processes that are statically distributed to a processor and access data in local memory only [12]. The shared memory model can be a better choice for application with extreme load imbalance.

The performance of two models mainly depends on relative cost of the communication provided by the hardware and the inherent load imbalance of the application [15]. As shared memory model has superior load balancing policies but poor data locality management, it is preferable to be implemented on machines where the communication is relatively cheap. In these cases, its advantage can overweight its limits.

On the contrast, message passing model offers better data locality properties but lack of load imbalance management, it favors to be implemented on machines where communication is relatively expensive. On these machines, its data locality advantage can be fully exploited to provide better performance even operating on the applications with extreme load imbalance.

Since it is widely believed that processor will be getting faster at higher rate compared to the memory and networks [15] [16], the relative cost of communication will also increase. So the message passing model will become an increasing efficient alternative in parallel programming future.

5. Implementation Method

This chapter will explain the implementation method adopted for the project. Firstly, the efforts concentrated on the operating systems implementation will be concluded. Then the implementation of the shared memory model and message passing model will be discussed.

5.1. Operating Systems Implementation

In order to create a operating systems as a platform for two parallel model implementation, there are several issue should be concerned, which involved cache coherency, virtual address and process management.

5.1.1. Cache Coherency Implementation

Since each processor holds its own level 1 cache in Cortex-A9MPCore. If one processor updates a new data to the shared memory address without notify other processors, the other processors can be operating on out of date data on its cache, which leads to inaccurate communication. So cache coherency is vitally significant in multiple processing systems.

Cache coherency means the changed of the data in processor cache should be visible to the other processors, which maintain the cache consistency in the communication between processors. There are mainly two approaches for cache coherency: snooping and updating [17].

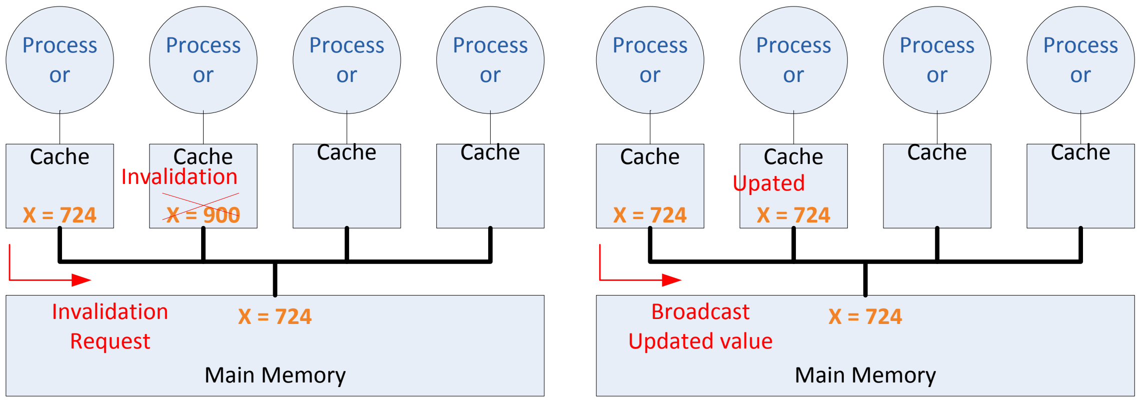


Fig 21 Cache Coherency Approaches

For the snooping approach, as all the processors continuously monitor the bus connecting the processors. If a processor update new value to a data item, all other copies of this data item in other processors' caches are configured as invalidated. This leads to a cache miss when the other processors attempt to fetch this data item from its own cache. And all the processors are forced to update new value for the data item from the main memory, which is illustrated in figure 21. While for the updating approach, if a processor updates a new value to a data item, it broadcasts its updated value to all the other processors to ensure the cache consistency.

Although both methods can maintain cache coherency between processors, snooping approach performs better under multiple writes to the same address space condition. It only require to invalid other caches for one write in snooping protocol, but have to broadcast multiple times in updating protocol.

The cache coherency in Cortex-A9MPCore is handled by a device known as Snoop Control Units (SCU) []. It maintain coherency between processors' L1 data caches with the implementation of MESI-based protocol as coherency management. In order to maintain coherency between processors, the configuration of specific register need to be marked as followed:

- The SCU is enabled, which control register located in the private memory region

- The processor access is configured to participate in the inner shareable domain

- The Memory Management Units (MMU) needs to be enabled.

- The page table need to be configured as "Normal", "Shareable", with a "write-back, write allocate" policy.

5.1.2. Virtual Address Implementation

In order to present better memory management and protection, virtual address is implemented in the operation system. This allows multiple processors operating on the same virtual address simultaneously while actually being stored in the different physical address.

By using page tables, Memory Manage Units (MMU) in Cortex-A9MPCore can translate virtual address to physical address. Page table is essential significant in virtual address implementation. It contains a series of entries which are organized by virtual address and it also provides access permission and memory attributes for memory translation.

The ARM MMU provides multi-level page table architecture with two levels of page table: level 1 (L1) and level 2 (L2) [17]. For the single level 1 page table, it is capable of dividing full 4GB physical address space into 4096 equally sized 1MB section. Therefore, the level 1 page table contains 4096 entries, which are all being word sized. Each entry can either hold a page table entry for translating a 1 MB section or a pointer to the base address of a level 2 page table [17].

In order to locate relevant entry in the page table, the most significant 12 bits will be provided to index the 4096 entry within the page table. And since the location of the page table of will be request for MMU to translate the virtual address to physical address. The base address of the page table, known as the translation table base address, is stored in a register in CP15, c2 [17].

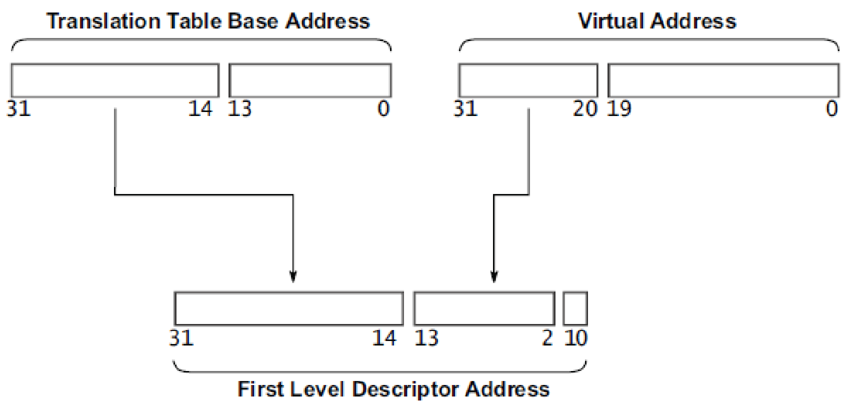

Translation Table Base Address
31
14
13
0
Virtual Address
31
20
19
0
31
14
13
2
1
0
First Level Descriptor Address

Fig 22 Finding the Address of the L1 Page Table Entry [17]

To illustrate the procedure of locating the address of the level 1 page table entry, a simple example will be offered. Assuming the level 1 page table is located at address 0x12300000 and the processor offers the virtual address 0x00200000. Since the top 12 bits of the virtual address will be accessed to index the location of the entries, 0x002 will be acquired in our case. And with each entry is word sized (4 bytes), the offset will be acquired by multiplying the entry index with entry size:

0x002 * 4 = 0x008 (Offset Address)

0x12300000 + 0x008 = 0x12300008 (Entry Address)

As the translation procedure explained above, the MMU will access the word from address 0x12300008 after receiving virtual address 0x00200000 from the processor.

Figure 23 shows that page table entry contains not only the physical base address for virtual address translating, but also access permission and memory attributes which hold the information required to access the corresponding physical address.

Finally, with the combination of the physical base address section from the page table entry and the lower 20 bits from the virtual address, the physical address can be generated by memory manage units, The whole procedure for L1 page table to generate physical address is illustrated in figure 23.

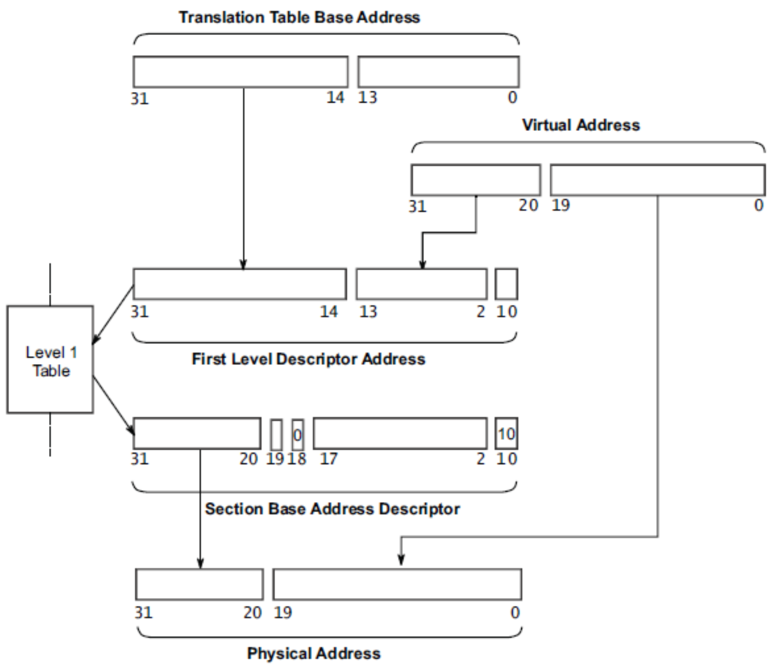


Fig 23 Generating Physical Address from the L1 Page Table [17]

When taking project purpose into consideration, virtual address will be required to guarantee the memory protection and memory coherence of the shared memory. Since both shared memory model and message model will be implemented on the operating system. A shared global address space will be provided to exchange the data between processors in shared memory model, while local memory will also be required for each processor involved to ensure the communication in message passing model.

Since each processor will holds its own page table which contains 4096 entries. By accessing multiprocessor affinity register in Cortex-A9MPCore, processor id can be obtained. And with left shift instruction, this processor id can be converted into 16KB offset for page table of each processor. When page table have been established, page table entries for each specific memory section can be stored in to ensure the translation of virtual address to physical address.

The memory map for virtual address and physical address translation in project is illustrated in figure 24. The shared data and code is located at the bottom of the memory map. As Memory coherent guarantee that the updated data from one processor can be seen by other processors and no processor operates on out of data cache, this section of the memory has to be configured as coherent since it can be accessed by different processors. The second section of virtual address space is acting like local memory for each processor, and is mapped to different physical address section. Due to the fact that it can only be accessed by its own processors, it can be configured as non coherent memory.

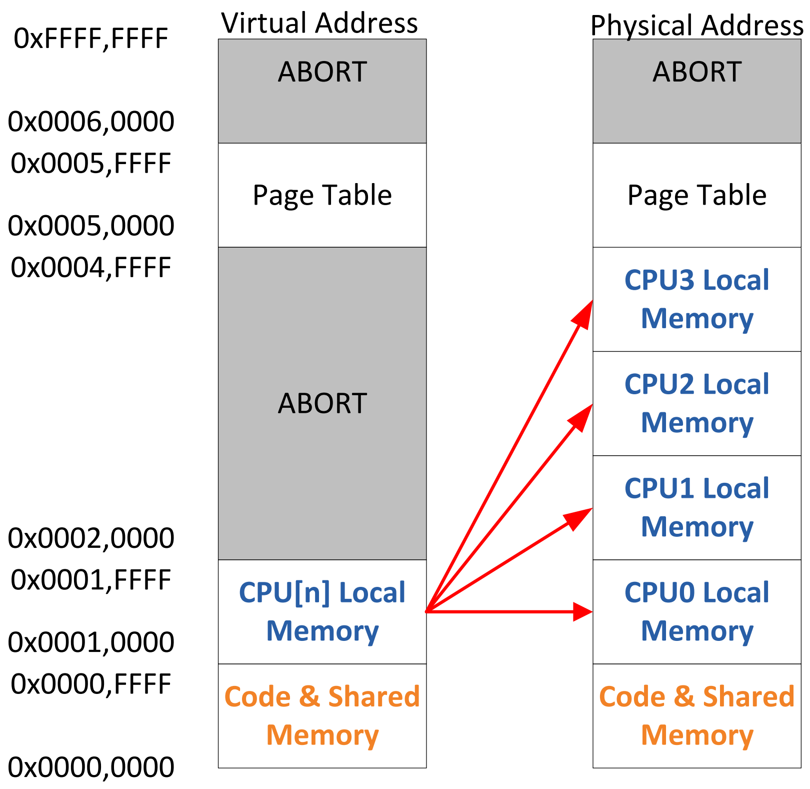


Fig 24 Virtual Address and Physical Address Translation

With virtual address implementation, the memory address becomes transparent to all processors expect the operating systems. In shared memory model, a global address space is allocated in shared memory section and can be equally accessed by all processors. While in message passing model, the processor cannot access shared memory section but have to passing message thought buffer dominated by operating systems.

5.1.3. Process Management Implementation

Process management can also be vitally important in operating systems. There are mainly three components in the process management, process creation, process destruction and process scheduling.

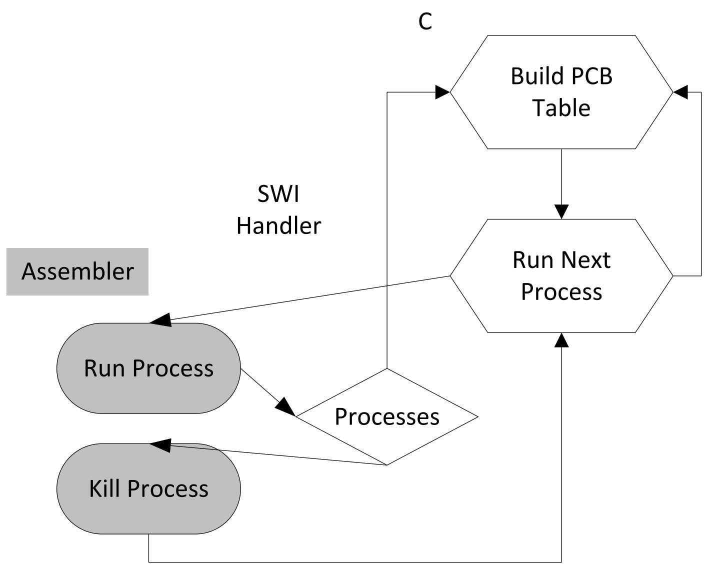


Fig 25 Process Management Implementation

Process control blocks (PCB) are implemented in the project to represent the state of the process. Since it is data structure that contains many piece of information related to a process such as program counter, CPU registers, process ID and other information. Linking all process control blocks together to form a larger data structure called PCB tables allows operating systems to keep track of all the processes under its control.

For the process creation, the operating system fetch the next process control block starting address from the PCB tables, and transfer it to the assembler function RunProcess through software interrupt handler. Based on the information provided by the process control block, RunProcess restore the state of the process and execute the process by adjusting the current program counter to the process program counter.

While for the process destruction, the operating system kills the process by calling RemoveProcess function. The function compare the current process ID with all the process ID to locate the process required to be removed and remove the corresponding process control block from the PCB tables.

The basic Round-robin scheduling is implemented in the project. The operating system executes the process control block in PCB tables in sequence.

As there are two processors involved in this project, each processor will generate its own PCB tables from the process in processes.c files. Although two processors will execute the same program, but will have different operations since the processors identification will be involved in the programming, which can be seen in the chapter 6.

5.2. Parallel Model Implementation

After the construction of the operating systems, two parallel programming model that the project concentrated on required to be implemented on the operating systems for the comparison work.

5.2.1. Shared Memory Model Implementation

In order to implement shared memory model, a global address space that can be equally accessed by each processors is already established in the virtual memory implementation. Cache coherency maintained by snoop control units ensure that the data modify in one processor’s cache can be visible by other processors’ cache, which guarantees the correct communication between processors. All these provide shared memory model implementation the basic platform.

However, an approach that guarantees the processor operating synchronously still need to be provided. And in order to prevent the processors from inappropriately accessing shared resources, lock and unlock operation will be required in shared memory model.

As an approach will be required for the processors to execute in correct order, binary semaphore can be one of the simplest approaches to achieve operating synchronously. By using a shared variable flag, binary semaphore provide a safety protocol to guarantee the parallel operations execute in the correct order.

Lock and unlock operation are also implemented in the project to prevent the shared global address space from inaccurate accessing. Four related functions are established to ensure the lock and unlock operation execute successfully.

In order to implemented lock and unlock operation, a function that initialize the lock operation is created at first. It simply accesses the lock variable address space and initializes the variable by writing a specific value to it. The specific value indicated the unlock condition of the lock variable, which in our project is marked with the value 0xFF.

After initialize the lock variable, the lock function will be provided. Since the shared address space may already be locked by other processors, the lock function firstly compares the value in lock variable with the specific unlocks value 0xFF mentioned before. If the variable appears to be locked, the lock function will go to standby condition and attempt to lock it later. If the variable is unlocked, the function will read the CPU id register in Cortex A series processors to obtain its processor identification and write it to the lock variable. This operation ensures the lock variable holds the identification of its locker which can be verified in unlocks function.

The unlock function is also vitally significant in lock and unlock operation. Since the fundamental idea of the lock and unlock operation is guarantee the variable can only be unlocked by its own locker. The unlock function require to access CPU id register to obtain its processor identification and compared it with the value stored in lock variable. If it is matched, the function goes to unlock operation which writes specific unlocks value 0xFF to the lock variable. However, the unlock operation will fail if the processor identification is not equal to the value stored in lock variable since the processor is not the lock owner.

Finally, a function that inspects the condition of the lock variable is also created. As it will return 0 in unlock condition while 1 in lock condition, the processors can utilize the function result as a flag to check whether it can access the shared variables controlled by locks.

With a global address space defined in virtual address implementation, binary semaphore and lock operations, the basic implementation of shared memory model is established.

5.2.2. *Message Passing Model Implementation*

For the implementation of message passing model, a global address space with cache coherency protocol is equally important since the buffer in message passing model need to be allocated in this area. As these are all already implemented in operating systems, constructing send and receive operation will be the major task for message passing implementation.

Unlike shared memory model, there is no shared variables exists. Process executed on the processors cannot access shared memory address directly, but have to transfer message through buffer controlled by the operating systems. This ensure the primitives in message passing model that data are transferred through buffer in global address space but are operated in local memory.

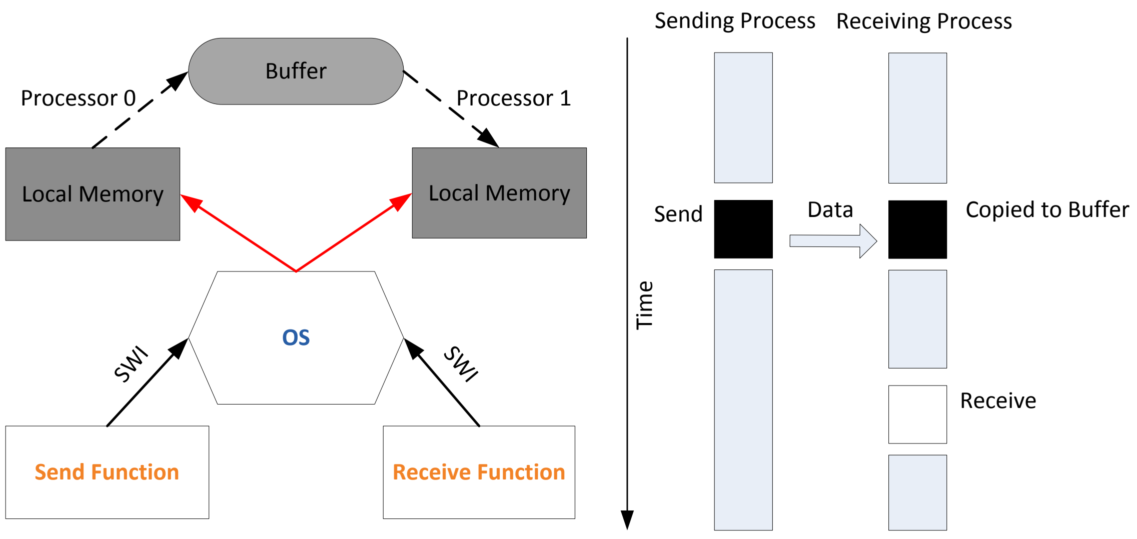


Fig 26 Basic Message Passing Implementation

The basic send function holds three parameters in its function, the destination which indicates the receiving processor for the communication, the message array which can hold the message transferred between processors and the length which state the length of the message. In order to pass message to the receiving processor, the processor utilize the software interrupt handler to enter operating systems and copy its message from local memory into the buffer. Related flags are constructed to ensure the synchronization in communication.

While for the basic receive operation, there are also three corresponding parameters in receive function, the source that indicates the sending processor, the message array and the length of the message. During the operation of the receive function, the receiving processor enter the operating systems by software interrupt handler and copy the corresponding message from the buffer to the local memory. Further operation can be implemented on the local memory instead of global address space.

In order to maintain the accuracy and synchronization in the communication, the send and receive function will verify the destination, source and length to ensure the message is transferred correctly. And by applying blocking communication, the send function will not return until the data have already been copied to the buffer while receive function will not return unless the data have already been transferred to the local memory.

With basic send and receive operation, the basic implementation of message passing model guarantee the processors communicate through passing message on shared memory hardware.

6. Result Analysis

6.1. Evaluation Algorithms

6.1.1. Quick sort

The quick sort algorithm has promising performance in its running time on an input array. In worst case, it makes $O(n^2)$ comparions which is rare in practice. The expected running time is actually $O(n\log n)$ with a small constant factors hidden in the $O(n\log n)$. Its advantage resides in the ability to sort in place. It is also worthy to mention that it works well even in virtual-memory environments [1].

Together with merge sort, they both apply the divide-and-conquer paradigm. In this section, the divide-and-conquer paradigm will first be introduced. The three steps for divide-and-conquer process for sorting array $A[p, \dots, r]$ are listed as follows [18]:

a) [Divide]: Partition the array $A[p, \dots, r]$ into two sub-arrays according to the array $A[q]$. The resulting sub-array $A[p, \dots, q-1]$ contains all the elements which is less than or equal to $A[q]$. On the contrary, all the elements in sub-array $A[q+1, \dots, r]$ are larger than or equal to $A[q]$.

b) [Conquer]: Sort the two sub-arrays $A[p, \dots, q-1]$ and $A[q+1, \dots, r]$ by calling to quicksort recursively.

c) [Combine]: Combine the sub-arrays to form the sorted array $A[p, \dots, r]$ finally.

Quick sort is indeed an algorithm which conforms to the paradigm strictly. The recursive procedure is shown in figure 27 below:

```
QUICKSORT($A, p, r$)
if $p < r$
    $q$ = PARTITION($A, p, r$)
    QUICKSORT($A, p, q - 1$)
    QUICKSORT($A, q + 1, r$)
```

Fig 27 Quick Sort Algorithms

Consequently, the sorting of the array can be sorted by the initial call as *QUICKSORT(A, 1, A.length)*. It is also obvious to notice the key step in the above algorithm is *PARTITION* procedure. It returns the index q for the divide part. Its algorithm is shown in figure 28 below. In *PARTITION* procedure, the last element in the array $A[r]$ is always selected as the pivot to partition the sub-array $A[p, \dots, r]$.

```
PARTITION($A, p, r$)
$x = A[r]$
$i = p - 1$
for $j = p$ to $r - 1$
    if $A[j] \leq x$
        $i = i + 1$
        exchange $A[i]$ with $A[j]$
exchange $A[i + 1]$ with $A[r]$
return $i + 1$
```

Fig 28 Partition algorithm

To summarize this algorithm, the performance of quick sort will be analyzed in worst-case partitioning and best-case partitioning. The running time of quick sort is totally depended on whether the partitioning is balanced or unbalanced. It means that the elements used for partitioning are crucial for the behavior of quick sort algorithm [18].

The worst-case for quick sort happens when the partitioning routine produces on sub-array with $n - 1$ elements and one with 0 elements. In this situation, the running time is $O(n^2)$. On another handm the best-case produces two sub-arrays with size no more than $n/2$. One is of size $\lfloor n/2 \rfloor$ and another is of size $\lceil n/2 \rceil$. The running time reduced impressively to $O(n \log n)$.

6.1.2. Merge sort

John von Neumann proposed merge sort in 1945. It is also an algorithm which follows the divide-conquer-combine paradigm closely.

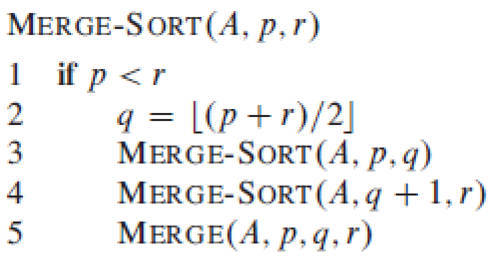
The three steps for divide-and-conquer process for sorting array are listed as follows [18]:

a) [Divide]: Divide the n-element sequence equally to be sorted into two sub-sequences of $n/2$ elements each.

b) [Conquer]: Sort the two sub-sequences recursively using merge sort.

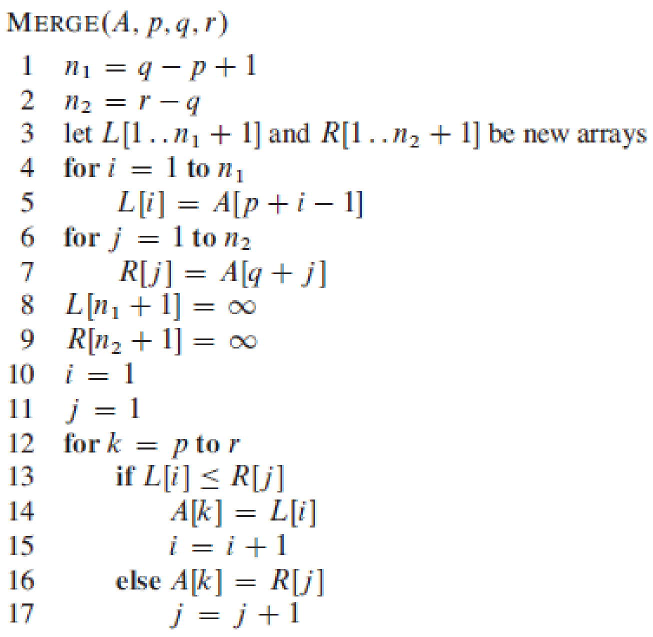
c) [Combine]: Merge the two sorted subsequences to produce the sorted answer.

The intuitive algorithm of merge sort is shown in figure 29. The sorting result of array $A[p, \ldots, r]$ can be obtained by calling *MERGE-SORT*$(A, 1, A.length)$.



```
MERGE-SORT($A, p, r$)
if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT($A, p, q$)
    MERGE-SORT($A, q + 1, r$)
    MERGE($A, p, q, r$)
```

Fig 29 Merge sort algorithm [18]



```
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 $j = 1$ 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 $L[i] \le R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
        $j = j + 1$
```

Fig 30 Merge algorithm [18]

Unlike quick sort mentioned above, the key step for merge sort relies on the combine step. This step is implemented by an auxiliary procedure called *MERGE*(A, p, q, r). r is an index in the array such that $p \le q < r$. This procedure will be given two sub-arrays $A[p, \ldots, q-1]$ and $A[q+1, \ldots, r]$ which is assumed to be sorted. It will return a single sorted array formed by combining the two sub-arrays. The pseudocode of this algorithm is illustrated in figure 29 where ∞ is a special symbol to indicate the ending.

6.2. Comparison Result

Since the implementation of the shared memory and message passing model have already been accomplished in chapter 5 section 2, and the evaluation algorithms quick sort and merge sort have been converted into parallel programming. The comparison result of the quick sort shows in table 1 while the result of the merge sort shows in table 2.

Quick Sort	Shared Memory Model	Message Passing Model
Input Size	Time (Cycles)	Time (Cycles)
5	4963	5764
10	6308	6371
15	6984	7317
25	9725	9404
50	14308	14809
100	21777	29187
250	52305	55479
500	112435	130969

Table 1 Quick Sort Comparison Result

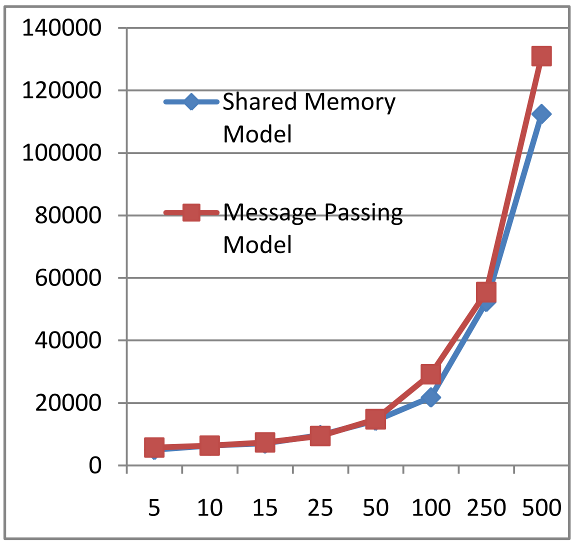
The results can be seen from the table 1 that there is no significant performance gap for quick sort implementation between two parallel programming models. This verifies the idea from the previous work on the comparison that message passing model can offer good performance even implemented on shared memory architecture hardware. However, although two models present approximately equal performance, shared memory model have slightly better performance in most of the cases.

When taking parallel programming merge sort into consideration, we can draw the same conclusion from the merge sort summarized in table 2. With no significant performance gap between two models, shared memory model performs slightly better under most of the conditions.

Merge Sort	Shared Memory Model	Message Passing Model
Input Size	Time (Cycles)	Time (Cycles)
5	4783	5525
10	6789	6921
15	7032	7453
25	9549	9823
50	14758	14882
100	22014	25692
250	53570	59814
500	124375	139061

Table 2 Merge Sort Comparison Result

Both results can be into plotted into graph which is illustrated in figure 31.



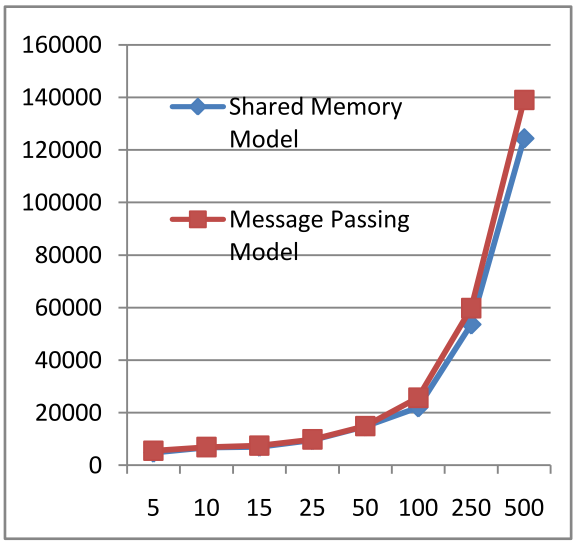


Fig 31 Sort Comparison Result

The graphs above illustrate the result that both models almost present the same performance but shared memory model have slightly better performance in most of the cases more clearly.

Previous work which discussed in chapter 4 indicates that message passing model will perform better on shared memory hardware than shared memory model without extreme load imbalance involved. However, in our project, the shared memory model has slightly better performance than message passing model.

This result can be caused mainly by four reasons, hardware supported cache coherency, lack of local memory, limited numbers of processors and complexity of the algorithms implemented.

In the shared memory model, all processors can equally access the global address space located in shared memory space and perform operation on it. While in message passing model, the processor transfer information through buffer but perform operation on local memory. So it consumes more time for shared memory model to maintain cache coherency than message passing model. However in Cortex-A9MPCore, there exists a snoop control units (SCU) dedicated in maintaining cache coherency in shared memory. This hardware supported cache coherency protocol reduces the performance gap between two models to maintain cache coherency.

Lack of local memory can be the primary reason for message passing model performs worse than expected. In Cortex-A9MPCore, there is only shared memory hardware architecture exists. Since the major advantage of message passing model is the ability to exploit data locality. Without local memory exists, the message passing model fail to benefit performance increase from its main advantage. Although we implemented virtual address to create local memory space for message passing model, it cannot match the benefits the local memory hardware provided.

Limited numbers of processors can also be a reason for message passing model to present worse performance than shared memory model. Although Coxtex-A9MPCore contains four cores nowadays, the ARM Workbench IDE, the project software tool, only support two core simulations. With the number of the processors increased, the shared memory model implementation will confront more bus traffic than message passing model implementation, which will decrease the shared memory model performance.

Finally, the algorithms implemented for the project may not be complicated enough. More complex algorithms can be implemented to present a more convincing result.

Since it is widely believed that processor will be getting faster at higher rate compared to the memory and networks, the relative cost of communication will also increase. And the core number will still increase in ARM processors. So the shared memory model will have slight advantage nowadays but the message passing model will become an increasing efficient alternative in parallel programming future.

7. Conclusion and Further Work

The project investigate the parallel efficiency of two widely use software implementation shared memory model and message passing model on conventional multi-core hardware Cortex-A9MPCore processor.

Shared memory model pass data through shared resource located in global memory. Advanced features like semaphore, barrier and lock/unlock function are created to ensure each processor performing synchronously without accessing shared resource improperly. While for message passing model, message and other communication parameters are packed and transfer between processors. With buffer and non-blocking communication, processor can operate asynchronously, which improve the efficiency of the message passing implementation.

Previous work on comparison between message passing and shared memory implementation indicates that both implementation hold its own advantages and limits. Data locality is strongly believed to be the advantage of the message passing model. With ability of fully exploit data locality, message passing implementation is capable of achieving better performance in most of the cases. However, shared memory holds the advantage of executing operations with extreme load imbalance because the inherent imbalance in the applications can be rearranged in the central queue and distributed evenly to the processes. Previous work also shows that the choice of the implementation will not be simply determined by the advantage of two implementations. Hardware architecture and features of the problems should also be taken into consideration.

In order to compare these two software implementations, a basic operating system consist of cache coherency, virtual address and process management implementation is constructed to provide a platform for two implementations. Then basic send/ receive operation are created for message passing implementation and lock/unlock operation are established for shared memory implementation. Finally two algorithms quick sort and merge sort are parallel programmed and utilize as a evaluation tool to obtain comparison result.

Project results indicates that when there is no local memory hardware architecture but hardware support cache coherency exits, the shared memory model can have slightly better performance than message passing model. Without local memory to exploit data locality, the message passing implementation performance will decrease. And hardware support cache coherency will reduce the performance gap between two implementations. All these lead to share memory model performs slightly better than message passing model in our project.

For the further work on the project, there are mainly four areas with improvement space. At first, scalability can be investigated if the software tool ARM provided support. Secondly, thread can be implemented in shared memory model to achieve better performance. Furthermore, application with inherent load imbalance should also be taken into consideration. And finally, more complex algorithms should be utilized to evaluate the performance of both models.

However, when taking future processor and memory development into consideration, it is widely believed that processor will be getting faster at higher rate compared to the memory and networks. And the cores number in processor will still increase in future. So the message passing implementation tends to obtain better performance in future.

Bibliography

[1] M. J. Flynn. Very high-speed computing systems. Proceedings of the IEEE. 1901–1909. 1966.

[2] C. Xavier, Sundararaja S. Iyengar. Introduction to Parallel Algorithms. Addison-Wesley Longman Publishing. 20-40. 1992

[3] Ananth Grama and Vipin Kumar. Introduction to Parallel Computing. Second Edition. Addison-Wesley Longman Publishing. 233-277. 2002

[4] Peter S. Pacheco. Parallel Programming with MPI. Morgan Kaufmann Publishers Inc. 11-65. 1996.

[5] Cortex-A9 Tutorial. http://www.arm.com/files/pdf/ARMCortexA-9Processors.pdf.

[6] Samuel H. Fuller and Lynette I. Millett. The Future of Computer Performance: Game Over or Next Level. 120-147. 2011.

[7] Blaise Barney. POSIX Threads Programming. 5-120. 2001.

[8] Jaroslaw Nieplocha and Robert J. Harrison and Richard J. Littlefield. Global arrays: a Portable "Shared-Memory" Programming model for Distributed Memory Computers. In *Proceedings of the 1994 conference on Supercomputing*. 340-350. 1994

[9] Calvin Lin and Lawrence Snyder. A Comparison of Programming Models for Shared Memory Multiprocessors. In Proceedings of the 1990 International Conference on Parallel Processing. 163-170. 1990

[10] Ton A. Ngo and Lawrence Snyder. On the Influence of Programming Models on Shared Memory Computer Performance. In Scalable High Performance Computing Conference (SHPCC '92). 1992

[11] Ton A. Ngo and Lawrence Snyder. Data Locality on Shared Memory Computer under Two Programming Models. 1993

[12] Tomas J. LeBlanc and Evangelos P. Markatos. Shared Memory Vs. Message Passing in Shared-Memory Multiprocessors. In Fourth IEEE Symposium on Parallel and Distributed Processing. 1992

[13] A. C. Klaiber and H. M. Levy. A comparison of message passing and shared memory architectures for data parallel programs. In *Proceedings of the 21st annual international symposium on Computer architecture* (ISCA '94). IEEE Computer Society Press, Los Alamitos, CA, USA, 94-105. 1994

[14] David Kranz and Kirk Johnson and Anant Agarwal and John Kubiatowicz and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory: Early Experience. In Fifth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPOPP). 54-63. 1993.

[15] Satish Chandra and James R. Larus and Anne Rogers. Where is Time Spent in Message-Passing and Shared-Memory Programs. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 61-73. 1994.

[16] Wong H. J. and Rendell A. P. Integrating software distributed shared memory and message passing programming. CLUSTER International Conference. 1-10. 2009

[17] Cortex-A Series Programmer Guider

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0013b/index.html.

[18] Thomas H. Cormen and Charles E. Leiserson and Ronald L. Rivest and Clifford Stein. Introduction to Algorithms. Third Edition. MIT Press. 171-180. 2009

APPENDIX : *Source Code*

PART 1   Virtual Memory Implementation

```
; -------------------
; Page tables
; -------------------
    LDR     r0, =||Image$$PAGETABLES$$ZI$$Base||
    MRC     p15, 0, r1, c0, c0, 5
    ANDS    r1, r1, #0x03
    MOV     r1, r1, LSL #14
    ADD     r0, r1, r0
    MOV     r2, #1024
    MOV     r1, r0
    MOV     r3, #0
    MOV     r4, #0
    MOV     r5, #0
    MOV     r6, #0
ttb_zero_loop
    STMIA    r1!, {r3-r6}
    SUBS    r2, r2, #1               ; Decrement counter
    BNE     ttb_zero_loop

    LDR     r1, =PABASE_VA0
    LDR     r2, =TTB_COHERENT
    ORR     r1, r1, r2
    STR     r1, [r0]

    MRC     p15, 0, r1, c0, c0, 5
    AND     r1, r1, #0x03
    MOV     r1, r1, LSL #20

    LDR     r3, =PABASE_VA1
    ADD     r1, r1, r3
    LDR     r2, =TTB_NONCOHERENT
    ORR     r1, r1, r2
    STR     r1, [r0, #4]

    MRC     p15, 4, r1, c15, c0, 0
    LSR     r1, r1, #20
    LSL     r2, r1, #2
    LSL     r1, r1, #20

    LDR     r3, =TTB_DEVICE
    ORR     r1, r1, r3
    STR     r1, [r0, r2]

    DSB
; ------------------
; Set location of level 1 page table
; ------------------
    MCR     p15, 0, r0, c2, c0 ,0
```

PART 2   Process Management Implementation

```
PCB *CreatPCB(int ProcessAddress)
{
    int i;
    PCB *p;
    p = (PCB*)malloc(sizeof(PCB));
    AID = AID + 1;
    ASP = ASP + 0x300;
    p->ProcessID = AID;
    p->ProcessState = 0;
    p->CPSR = 0x10;
    for (i = 0; i< 12; i++){
        p->R[i] = 0;
    }
    p->SP = ASP;
    p->LR =    0;
    p->PC = ProcessAddress;
    p->Before = NULL;
    p->Next = NULL;
    return p;
}
void AddAsmProcess(int *TableAddress, int TotalNumber)
{
    int ProcessAddress;

    ProcessAddress = *TableAddress;
    Start = CreatPCB(ProcessAddress);
    Current1 = Current2 = Start;

    while( AID < (TotalNumber-1) ){
        ProcessAddress = *(++TableAddress);
        Current1->Next = CreatPCB(ProcessAddress);
        Current2 = Current1->Next;
        Current2->Before = Current1;
        Current1 = Current2;
    }
}

void RemoveProcess(int ProcessID)
{
    PCB *Remove, *RemovePre, *RemoveNx ;
    Remove = Start;
    while ( Remove->ProcessID != ProcessID){
        Remove = Remove->Next;
    }
    if (Remove->Before == NULL){
            Start = Remove->Next;
            Start->Before = NULL;
    }
    else{
        RemovePre = Remove->Before;
        RemoveNx = Remove->Next;
        RemovePre->Next = RemoveNx;
        RemoveNx->Before = RemovePre;
    }
    free(Remove);

}

KillProcess
    LDMFD   sp!, {r0-r12,lr}
    LDMFD   sp!, {r4}
    LDR     r0, =Kill
    BL      PrintString
    MOV     r0, r4
    BL      PrintDec
    LDR     r0, =NewLine
    BL      PrintString
    MOV     r0, r4
    BL      RemoveProcess
    LDMFD   sp, {r13,r14}^; Load r13&r14 in UseMode
    NOP
    ADD     sp, sp, #8
    LDMFD   sp!, {r5}
    MSR     SPSR_cxsf, r5; Load State Register
    LDMFD   sp!, {r0-r12,pc}^

;-----------------------------------
;----------------------------------
RunProcess
    MRS     r5, SPSR
    STMFD   sp!, {r5}
    STMFD   sp, {r13,r14}^
    NOP
    SUB     sp, sp, #8
    LDR     r0, =NewLine
    BL      PrintString
    LDMFD   r8!,{r4-r6}
    STMFD   sp!, {r4}; Store ProcessID
    ADDS    r5, r5, #0
    BLNE    Continue
    LDR     r0, =Begin
    BL      PrintString
    MOV     r0, r4
    BL      PrintDec
    MSR     CPSR_cxsf, r6
    MOV     r0, r8
    LDMFD   r0!,{r1-r15}
```

PART 3 Shared Memory Implementation

```
;-----------------------------------
;-----------------------------------
InitLock

    MOV     r1, #0xFF; Unlock
    STR     r1, [r0]
    LDMFD   sp!, {r0-r12,pc}^
```

```
;-----------------------------------
;-----------------------------------
Lock
    LDR     r1, [r0]
    CMP     r1, #0xFF
    BNE     Lock

    MRC     p15, 0, r1, c0, c0, 5
    AND     r1, r1, #0x03
    STR     r1, [r0]
    LDMFD   sp!, {r0-r12,pc}^
;-----------------------------------
;-----------------------------------
UnLock
    MRC     p15, 0, r1, c0, c0, 5
    AND     r1, r1, #0x03
    LDR     r2, [r0]
    CMP     r1, r2
    BNE     Unlock
    MOV     r1, #0xFF
    STR     r1, [r0]
Unlock
    LDMFD   sp!, {r0-r12,pc}^

;-----------------------------------
;-----------------------------------
TestLock
    LDR     r1, [r0]
    LDMFD   sp!, {r0}
    CMP     r1, #0xFF
    MOVEQ    r0, #0x0; UnLock return 0
    MOVNE    r0, #0x1; Lock return 1
    LDMFD   sp!, {r1-r12,pc}^

;-----------------------------------
;-----------------------------------
```

PART 4 Message Passing Implementation

```
;-----------------------------------
;-----------------------------------
```

```
SendMessage
    MRC     p15, 0, r0, c0, c0, 5; Read CPU ID register
    AND     r0, r0, #0x03
    MOV     r5, r0
    LDMFD   sp!, {r0-r2}
    LDR     r4, =0x000FA000; Buffer Location
    ADD     r4, r4, #4
    STR     r5, [r4]
    ADD     r4, r4, #4
    STR     r0, [r4]
    ADD     r4, r4, #4
    STR     r2, [r4]
    ADD     r4, r4, #4

smloop
    LDR     r3, [r1]
    ADD     r1, r1, #4
    STR     r3, [r4]
    ADD     r4, r4, #4
    SUBS    r2, r2, #1
    BNE     smloop

    MOV     r9, r4
    MOV     r0, #1;Send Blocked Flag
    STR     r0, [r9]
    LDR     r4, =0x000FA000
    MOV     r7, #1;Receive Blocked Flag
    STR     r7, [r4]

sblocked
    LDR     r0, [r9]
    CMP     r0, #0
    BNE     sblocked

    LDMFD   sp!, {r3-r12, pc}^

;-----------------------------------
;-----------------------------------
ReceiveMessage
    MRC     p15, 0, r0, c0, c0, 5; Read CPU ID register
    AND     r0, r0, #0x03
    MOV     r5, r0
    LDMFD   sp!, {r0-r2}
    LDR     r4, =0x000FA000; Buffer Location

rblocked
    LDR     r9, [r4]
    CMP     r9, #1;Blocked Flag
    BNE     rblocked

    MOV     r9, #0
    STR     r9, [r4]

    ADD     r4, r4, #4
    LDR     r6, [r4]
    CMP     r6, r0
    BNE     WrongMessage
    ADD     r4, r4, #4
    LDR     r7, [r4]
    CMP     r5, r7
    BNE     WrongMessage
    ADD     r4, r4, #4
    LDR     r8, [r4]
    CMP     r2, r8
    BNE     WrongMessage
    ADD     r4, r4, #4

rmloop
    LDR     r3, [r4]
    ADD     r4, r4, #4
    STR     r3, [r1]
    ADD     r1, r1, #4
    SUBS    r2, r2, #1
    BNE     rmloop

    MOV     r0, #0
    STR     r0, [r4]

    LDMFD   sp!, {r3-r12, pc}^
```

PART 5 Sorting Parallel Programming

```
void Processc1(void)
```

```
{
    int num;
    int key,i,j,k,a;
    int ID;

    ID = Cpu_ID();

    if (ID == 0){

        Init_Lock(&pivotlock);
        Lock(&pivotlock);

        for(num = 0; num < MAXNUMBER; num++ ){
            sort[num] = (rand()%50);
        }

    printf("Before sorting is:\n");
    Printlist(sort,MAXNUMBER);
    printf("\n");

        k = choose_pivot(0,MAXNUMBER-1);
        swap(&sort[0],&sort[k]);
        key = sort[0];
        i = 1;
        j = MAXNUMBER-1;
        while(i <= j){
            while((i <= MAXNUMBER-1) && (sort[i] <= key))
                i++;
            while((j >= 0) && (sort[j] > key))
                j--;
            if( i < j)
                swap(&sort[i],&sort[j]);
        }
        swap(&sort[0],&sort[j]);

        pivot = j;

        UnLock(&pivotlock);

        Quicksort(sort,0,j);
    printf("After sorting is:\n");
    Printlist(sort,MAXNUMBER);
    printf("\n");
    }
    else{
        a = Test_Lock(&pivotlock);
        while (a != 0) {
            a = Test_Lock(&pivotlock);
        }

        Quicksort(sort,pivot,MAXNUMBER-1);
    }
    KillProcess();
}


void Processc1(void)
{
    int sort[MAXNUMBER];
    int length[MAXNUMBER];
    int num;
    int key,i,j,k;
    int ID;
    ID = Cpu_ID();

    if (ID == 0){
        for(num = 0; num < MAXNUMBER; num++ ){
            sort[num] = (rand()%50);
        }

    printf("Before sorting is:\n");
    Printlist(sort,MAXNUMBER);
    printf("\n");

        k = choose_pivot(0,MAXNUMBER-1);
        swap(&sort[0],&sort[k]);
        key = sort[0];
        i = 1;
        j = MAXNUMBER-1;
        while(i <= j){
            while((i <= MAXNUMBER-1) && (sort[i] <= key))
                i++;
            while((j >= 0) && (sort[j] > key))
                j--;
            if( i < j)
                swap(&sort[i],&sort[j]);
        }
        swap(&sort[0],&sort[j]);

        length[0] = MAXNUMBER-j;

        Send_MP (0x1, length, 1);

        Send_MP (0x1, &sort[j], length[0]);

        Quicksort(sort,0,j-1);

        Rece_MP (0x1, &sort[j], length[0]);

    printf("After sorting is:\n");
    Printlist(sort,MAXNUMBER);
    printf("\n");

    }
    else{
        Rece_MP (0x0, length, 1);
        Rece_MP (0x0, sort, length[0]);
        Quicksort(sort,0,(length[0]-1));
        Send_MP (0x0, sort, length[0]);
    }
    KillProcess();
}

//Quick Sort Function//
void Printlist(int sort[],int n)
{
   int i;
   for(i=0;i<n;i++){
     printf("%d\t",sort[i]);
     if ((i != 0)&&((i%50) == 0)){
     printf("\n");
```

```
      }
   }
}

void swap(int *x,int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

int choose_pivot(int i,int j )
{
    return((i+j) /2);
}

void Quicksort(int sort[],int m,int n)
    {
       int key,i,j,k;
       if( m < n)
       {
          k = choose_pivot(m,n);
          swap(&sort[m],&sort[k]);
          key = sort[m];
          i = m+1;
          j = n;
          while(i <= j)
          {
             while((i <= n) && (sort[i] <= key))
                   i++;
             while((j >= m) && (sort[j] > key))
                   j--;
             if( i < j)
                   swap(&sort[i],&sort[j]);
          }
          swap(&sort[m],&sort[j]);
          Quicksort(sort,m,j-1);
          Quicksort(sort,j+1,n);
       }
    }
```