August 2014
Master’s Degree Thesis

# A Feasibility Test of MPI Applications over Microkernel for Many Cores

Graduate School of Chosun University

Department of Computer Engineering

Irvanda Kurniadi Virdaus

# A Feasibility Test of MPI Applications over Microkernel for Many Cores

매니코어용 마이크로 커널 상의 MPI
응용프로그램 성능 평가

August 25, 2014

Graduate School of Chosun University

Department of Computer Engineering

Irvanda Kurniadi Virdaus

# A Feasibility Test of MPI Applications over Microkernel for Many Cores

Advisor: Prof. Moonsoo Kang, PhD

A thesis submitted in partial fulfillment of the requirements for a Master's degree

April 2014

Graduate School of Chosun University

Department of Computer Engineering

Irvanda Kurniadi Virdaus

# 쿠르니아디 비르다우스 이르반다의
# 석사학위논문을 인준함

2014 년 5 월

조선대학교 대학원

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# ABSTRACT

A Feasibility Test of MPI Applications over Microkernel for Many Cores

Irvanda Kurnadi Virdaus
Advisor: Prof. Moonsoo Kang, Ph.D.
Department of Computer Engineering
Graduate School of Chosun University

The multi-core architecture term appears due to the limitation of the single-core architecture which the improvement of clock frequency is difficult to be gained. A multi-core processor is typically made by two or four independent processor cores in a single chip which are connected through an on-chip bus. Because of the number of cores are increasing from 4 to 8, 16, and so on, a many-core term is appeared. Therefore a system which uses many-core architecture is supposed to be highly parallel. A parallel program makes a task into threads or processes which it should be spread across multiple cores. In microkernel-based systems, the communication of each process is assisted by Inter Process Communication (IPC). The message-passing allows the processes to communicate with each other using IPC. The parallel programming that uses message passing standard is called Message Passing Interface (MPI).

In this thesis, MPI library is ported to the L4Linux on top of L4/Fiasco microkernel over many-core architecture. The MPICH is chosen to be used as the MPI library routines. The performance comparisons from three different scenarios have been done to check the feasibility of MPI application on microkernel OS. From the benchmarking application, we observed how MPI library takes influence in the communication between processes which occurs in the many-core system. The performance improvement is shown by the matrix multiplication test program that proves the feasibility of MPI on microkernel OS over many-core system.

# 한 글 요 약

## 매니코어용 마이크로 커널 상의 MPI 응용프로그램 성능 평가

쿠르니아디 비르다우스 이르반다
지도 교수: 강문수
컴퓨터공학과
대학원, 조선대학교

본 논문은 매니 코어 시스템 상에서 마이크로 커널을 사용하는 환경에서 매니 코어의 성능을 효율적으로 사용하기 위해 MPI (Message Passing Interface) 기반의 병렬 응용 프로그램을 수행하기 위한 시스템을 구축하고, 그 성능을 평가하였다. 프로그램 연산 성능을 더욱 향상시키기 위해 근래의 CPU는 단일 코어가 아닌, 4, 8, 16, 32개의 복수의 코어를 내부에 장착하여 한번에 하나의 프로그램만 실행할 수 있는 것이 아니라 매니 코어의 개수만큼 동시에 복수 개의 프로그램을 실행할 수 있다. 그러나 이러한 하드웨어적인 특성에도 불구하고 이러한 자원을 효율적으로 사용하여 병렬 연산을 수행할 수 있는 병렬 연산 플랫폼 또는 병렬 프로그래밍 기법이 thread 기반의 단순한 형태로 머물러 있다. 또 멀티 코어 시스템에서 일반적으로 사용되는 리눅스와 같은 모노리틱 커널은 하드웨어와 운영체제의 각 기능을 하나로 묶어 빠른 성능을 보이지만, 운영체제 일부의 부분에 문제가 발생할 경우, 시스템 전체가 멈추는 불안정성을 내포하고 있다. 마이크로 커널은 하드웨어를 운영하는 최소의 부분만을 운영체제가 관리하고, 기존의 운영체제에 포함되어 있는 메모리관리, 네트워크 관리 등이 사용자 프로세스화 되어 있어 일부의 문제점이 전체의 문제점으로

확장되는 것을 최소화할 수 있어 수행속도에서는 모노리틱 커널에 비해 다소 떨어지나 안전성에서 모노리틱 커널에 비해 우수한 것으로 알려져 있다. 안정적이며 효율적인 병렬 연산 시스템을 구축하기 위해 본 논문은 병렬 연산에서 가장 많이 사용되는 병렬 연산 플랫폼인 MPI를 마이크로 커널을 사용하는 멀티코어 시스템용으로 구축하고, 모노리틱 커널을 사용하는 멀티코어 시스템과 성능을 비교 분석하였다. 벤치 마크, 병렬 행렬 연산, 이미지 병렬 처리 응용 프로그램들의 수행 속도를 측정하여 성능 비교 결과, 기존의 모노리틱 커널 멀티 코어 시스템에 비해 수행 속도가 다소 떨어지는 점을 확인할 수 있었으나 그 정도가 미미함을 관찰할 수 있었다. 그러나 전체 시스템의 안정성이 크게 증가됨으로 전체적인 시스템의 신뢰도는 마이크로 커널 멀티 코어 시스템이 우수함을 확인하였다. 또, 싱글호스트 내에서 매니 코어들간의 MPI 통신은 수행 성능의 제약이 MPI 프로세스들 간의 공유메모리 접근 방법에 크게 영향을 받는다는 것을 확인할 수 있었다.

# I. INTRODUCTION

Many-core processor system is something that is developed to overcome the limitation of single-core which is hard to improve the clock frequencies. Many-core processor system arise from the multi-core term which refers to the processor which is typically made by two until eight independent processor cores in single chip connected through an on-chip bus. The many-core term just exceed beyond the number of cores in multi-core system.

## A. Problem Statement

Even though the development of single processor has rapidly increased, but the way of serial computing perform is quite slow in term of improvement. Moreover the large amounts of memory are not accessible by a single processor. Therefore, the multi-core system is introduced to implement the parallel computing to overcome the limitation of single-core and serial computing.

Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller instructions, which are then solved concurrently. There are several reasons why use parallel computing, such as save time, solve larger problems, provide concurrency, use of non-local resources, and overcome the serial computing limitation. By using parallelization in many-core processor system, the memory access resources become more expensive.

## B. Research Objectives

In parallel computing, there are several concepts of parallel programming models which are related to the topic in this thesis. There are shared memory model (without threads), shared memory model (with threads), and distributed memory model. Differences between shared memory model and distributed memory model will be discussed in the next chapter. However, the using of parallelization in many-core processor system cause the memory access resource becomes expensive. Hence, the distribution memory model of parallel programming can overcome this issue by implementing the message passing interface (MPI).

The microkernel OS is chosen to be implemented along with MPI library over many-core processor system because it has the important role of IPC. This IPC is used to pass message between CPUs. The other reason of using the microkernel OS is the scalability of microkernel which can be implemented into tiny embedded system.

The objective of this thesis is quite applicative in term of contribution. MPI library will be ported to microkernel OS in order to run the MPI-based image processing application. This application will be observed and measured to be compared with the performance obtained in other system.

## C. Thesis Layout

The rest of the thesis is organized as follows. First, in Chapter II, we give a brief explanation about background concepts including many-cores system, parallel computing, Message Passing Interface (MPI), Microkernel, and the L4 microkernel family. In Chapter III, we give a brief explanation about various implementation of MPI in several systems. In Chapter IV, the implementation of MPI on Fiasco.OC is explained. In Chapter V, the performance evaluation is evaluated and discussed. Finally, the conclusions of the thesis are given in Section VI.

## II. BACKGROUND CONCEPT

This section will provide background information about the architectural design of many-core processor system, the model of parallel programming, and the L4 family of Microkernel Operating System.

The predictions of transistor growth have been discussed since a long time ago and yet we have to face the fact that the use of transistors in a single chip processor is hardly optimize do to the improvement of frequency in a single chip. One of the famous laws is Moore’s Law which states the number of transistors doubling approximately every two years. Figure 1 shows how the clock frequency remains fairly constant since 2010.

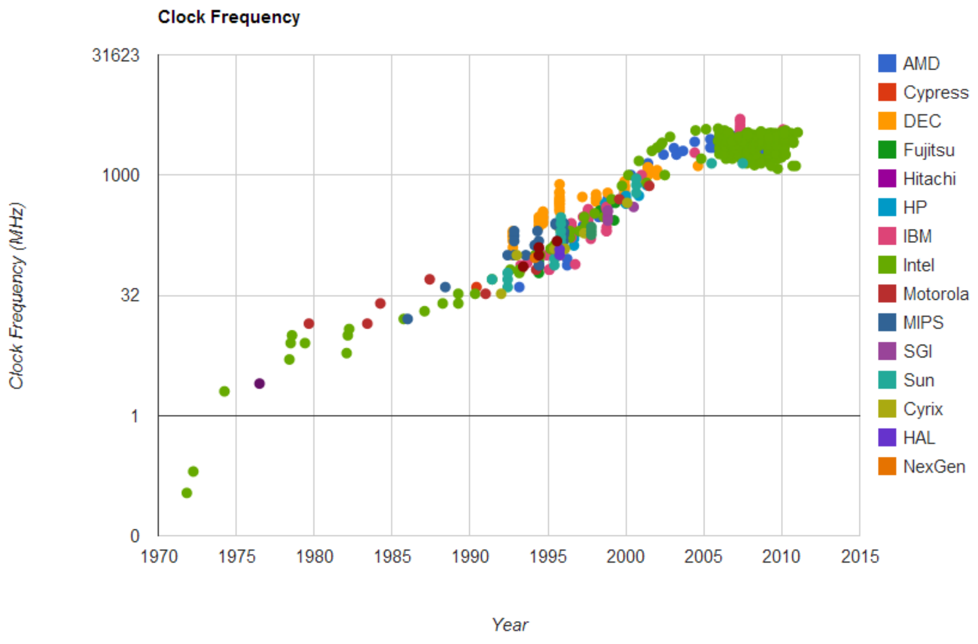


*Figure 1. The growth of clock frequency every year*

The increasing number of transistor in a single chip is getting riskier to be realized. The increasing value of clock frequency will generate a lot heat in return. Instead of increasing the clock frequency, the concept of getting faster processor shifted to create the multiple cores in a single processor that we call nowadays as multi-core system. The concept of multi-core system is also developing rapidly to a huge number of cores in a single processor, which we call as a many-core system.

## A. Many-Core System

A multi-core processor is a single computing component with two or more independent actual central processing units (called "cores"), which are the units that read and execute program instructions. The instructions are ordinary CPU instructions such as add, move data, and branch, but the multiple cores can run multiple instructions at the same time, increasing overall speed for programs amenable to parallel computing. Manufacturers typically integrate the cores onto a single integrated circuit die (known as a chip multiprocessor or CMP), or onto multiple dies in a single chip package.

Because of the number of cores is rising coupled from 2 or 4 to 8, 16, or many more, the term many-core created. A many-core system, despite the similarity with multi-core system, introduced new challenges. The current paradigm that we have on Operating Systems realized because of the characteristic of single-core; multiplex the CPU. However, in a many-core system we will have much more than one core and possibly multiplexing CPU is not necessary. Therefore the advent of new paradigm is needed.

## B. Parallel Computing

Traditionally, computer software has been written for serial computation. The way of serial computing execute the program is by constructed and implemented as a serial stream of instructions. These instructions are executed on a central processing unit on one computer. Only one instruction may execute at a time after that instruction is finished, and then the next is executed. Parallel computing, on the other hand, uses multiple processing elements simultaneously to solve a problem. This is accomplished by breaking the problem into independent parts so that each processing element can execute its part of the algorithm simultaneously with the others. The processing elements can be diverse and include resources such as a single computer with multiple processors, several networked computers, specialized hardware, or any combination of the above.

The concept of parallelism is appeared to solve development problem in increasing the clock frequency in single-core system. Therefore the concept of parallelism arises concurrently with the development of multi-core system. A multi-core processor is a processor that includes multiple execution units ("cores") on the same chip. These processors differ from superscalar processors, which can issue multiple instructions per cycle from one instruction stream (thread); in contrast, a multi-core processor can issue multiple instructions per cycle from multiple instruction streams. The term of many-core system is used to describe the multi-core system in High Performance Computing platform that contains more than 8 to 16 cores for each processor (single chip).

## C. Message Passing Interface (MPI)

Message Passing Interface (MPI) is a standardized and portable message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in Fortran or the C programming language. There are several well-tested and efficient implementations of MPI, including some that are free or in the public domain. These fostered the development of a parallel software industry, and there encouraged development of portable and scalable large-scale parallel applications.

In the beginning of the 1990s, a great variety of message-passing platforms for high performance computing existed. The message-passing model had already proofed its usefulness for high performance computing. It was in 1992 when at the Workshop on Standards for Message-Passing in a Distributed Memory Environment the developers of the most widespread message-passing platforms met to identify essential features for a standard message-passing interface. This first version of the standard consisted of mechanisms for point-to-point communication (send/receive), collective communication (broadcast, scatter, reduce, etc.), user defined data types; groups, contexts and communicators to ease writing portable parallel libraries, process topologies to aid mapping of processes to hardware benefiting locality, environment management (mainly for error handling and MPI-application attributes), a profiling interface and language bindings for C and Fortran.

To start MPI applications, it is necessary to setup a running environment, start processes on a specified processor of a node and to setup basic control channels among the process manager and the spawned processes. This is handled by process managers. MPICH uses the Process Management Interface (PMI) as a wire-protocol. It enables processes, which were started by the process manager, to send control messages in a standardized way. In a POSIX environment the process manager will inherit unidirectional pipes and a bidirectional PMI command socket into the newly fork( )ed process. The unidirectional pipes redirect stdout and stderr to the process manager for labeling (rank@node> my msg). The PMI command descriptor is then either made known using the PMI_FD environment variable or acquired by connecting to a hostname:port-tuple passed in the PMI_PORT environment variable. PMI control commands include set/get for a key value store, barrier synchronization, dynamic process management, general information about the processes (e.g. rank or number of processes) and the initialize/finalize commands.

## D. Microkernel

Microkernel is one kind of kernel; which in computing, the kernel is a computer program that manages input/output requests from software and translates them into data processing instructions for the central processing unit and other electronic components of a computer. The kernel is a fundamental part of a modern computer's operating system.

In computer science, a microkernel (also known as μ-kernel or Samuel kernel) is the near-minimum amount of software that can provide the mechanisms needed to implement an operating system (OS). These mechanisms include low-level address space management, thread management, and inter-process communication (IPC). If the hardware provides multiple rings or CPU modes, the microkernel is the only software executing at the most privileged level (generally referred to as supervisor or kernel mode). Traditional operating system functions, such as device drivers, protocol stacks and files systems, are removed from the microkernel to run in user space. In source code size, microkernels tend to be under 10,000 lines of code, as a general rule. MINIX's kernel, for example has fewer than 6,000 lines of code.

When a computer program (in this case called a process) makes requests of the kernel, the request is called a system call. Various kernel designs differ in how they manage system calls (time-sharing) and resources. For example, a monolithic kernel executes all the operating system instructions in the same address space to improve the performance of the system. A microkernel runs most of the operating system's background process in user space, to make the operating system more modular and, therefore, easier to maintain. This comparison of monolithic kernel and microkernel is shown in figure 2.

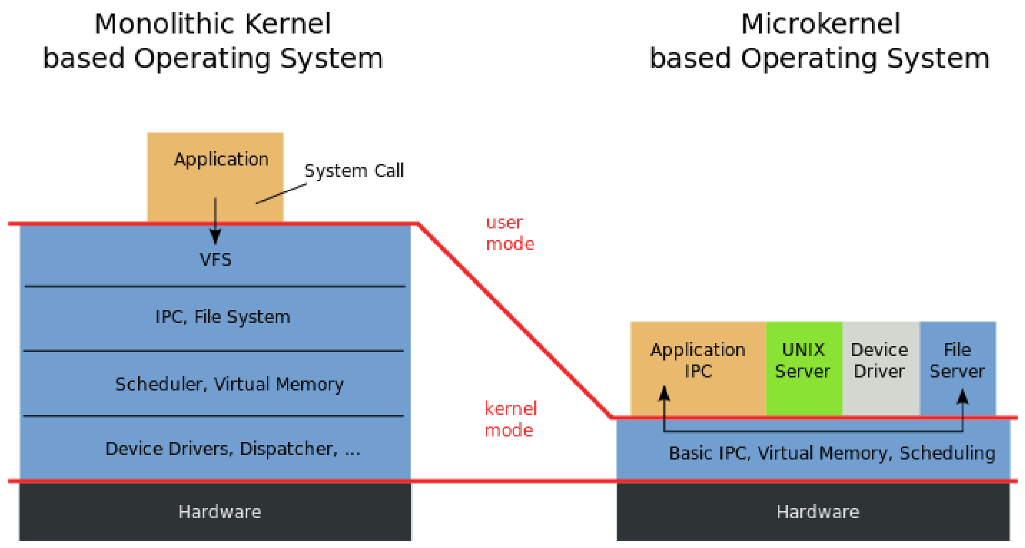

Monolithic Kernel based Operating System
Microkernel based Operating System
Application
System Call
VFS
IPC, File System
Scheduler, Virtual Memory
Device Drivers, Dispatcher, ...
Hardware
user mode
kernel mode
Application IPC
UNIX Server
Device Driver
File Server
Basic IPC, Virtual Memory, Scheduling
Hardware

***Figure 2. Monolitic kernel vs microkernel***

The advantages of microkernel are it is more robust, more reliable and it can provide real time operating systems for users. However, the monolithic kernel architecture is more powerful and superior compared to the microkernel and this make an obvious advantage for monolithic kernel. This comparison was demonstrated to show the poor performance of the earlier example of the microkernel such as Mach and Chorus OS. Nevertheless, the latest generation of microkernel has successfully shown the improvement to reduce the performance gap between these architectures.

## E. The L4 Microkernel Family

L4 is a family of second-generation microkernels, generally used to implement Unix-like operating systems, but also used in a variety of other systems.

### 1. L4/Fiasco

The L4 microkernel is one of the prominent microkernels in the research and academic community. It was created by Jochen Liedtke, which is the successor of the L3 microkernel, to address the poor performance of the earlier microkernels. L3 proved that microkernels can be fast, if performance critical aspects are optimized. In practice improving performance means that IPC among microkernel servers must either be avoided or fast. The L3 microkernel showed that IPC can be heavily optimized to achieve performance improvements by a factor of 22 compared with the Mach microkernel. He overcame the inaccurate concepts of Mach and simplified the concepts. The main concept of L4 is that a microkernel does no real work as it only provides inevitable mechanisms and no policies implemented.

L4 and L3 both retained scheduling in the kernel for performance reasons. Virtual memory management is separated between the kernel (for CPU exception handling) and a user-level pager that maps pages according to the policy implemented by the region manager. All IPC in kernel is synchronous. The original L4 application programmers interface (API) knew only 7 “syscalls” in total: IPC (for data transfer, interrupt handling, timeouts and mapping or granting pages), unmapping of flexpages from other tasks, task creation, thread creation and execution, thread scheduler to set the priority, time-slice or preempter of a thread, thread switching for yielding the calling thread or running another, and id_nearest returns the first IPC peer on the path to destination.

Fiasco, in its current version named Fiasco.OC (Fiasco Object Capability) is a descendant of L4. This kernel, written in C, C++, and Assembly, introduced an object-oriented API where kernel objects correspond to user-level objects. Five main kernel objects exist: task, thread, IPC-Gate, Factory, and IRQ. Other kernel objects are the Scheduler (set priority, CPU or timeslice), Icu (access to the Interrupt controller) and Vlog that implements a rudimentary console. All capabilities can be mapped or granted into tasks as well as unmapped from tasks.

## 2. L4Re

The L4 Runtime Environment (L4Re) is the user-land for the Fiasco.OC kernel. Together L4Re and Fiasco.OC constitute an object-capability system. Every IPC-Gate belongs to a user-level object. L4Re has four main user-level objects: Dataspaces are abstract memory (e.g. RAM, disk, memory-mapped I/O regions) and can also be shared among tasks (e.g. shared-memory). The Memory Allocator makes anonymous memory available using data-spaces. A Region Manager handles the address space of a task. This is achieved by assigning a Dataspace to virtual memory and thereby making it visible to the task. It is implemented in the task using a map of (memory region, data-space manager) tuples. When a page-fault occurs, the kernel issues a page-fault IPC to the Dataspace Manager (e.g. a user-level pager) that is responsible for this virtual memory region. Namespace objects are used to associate names with capabilities. Namespaces can be nested and are used to implement access control mechanisms. Access to user-level objects is announced by linking user-level object capabilities into the Namespace of the application. Another advantage of this approach is the possibility to substitute user-level objects that share a compatible interface without any changes to the application.

Fiasco.OC is not a full Operating System. There is neither a device driver nor the file system. Therefore Fiasco.OC needs this L4Re to handle the runtime environment by adding services as user-level components. The component diagram of L4Re is shown in figure 3.

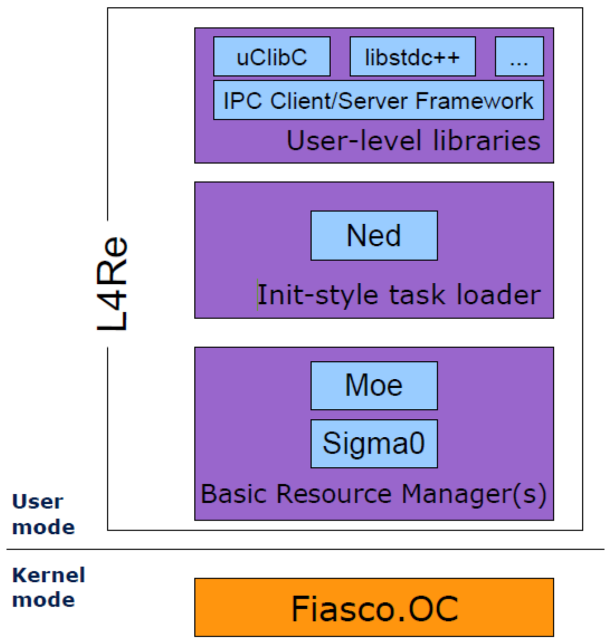


***Figure 3. L4 runtime environment***

## 3. L4Linux

L4Linux were ported to Linux kernel by treating it like another port of Linux to a new architecture. Currently, L4Linux has been updated to the Linux Kernel 3.8. The design overview for the L4Linux is depicted in the following figure 4. L4Linux can be run side by side with other real-time operating systems on top of L4/Fiasco. The real time operating system's proper operation will not be violated as L4Linux is run on user mode and cannot override the operation of L4/Fiasco.

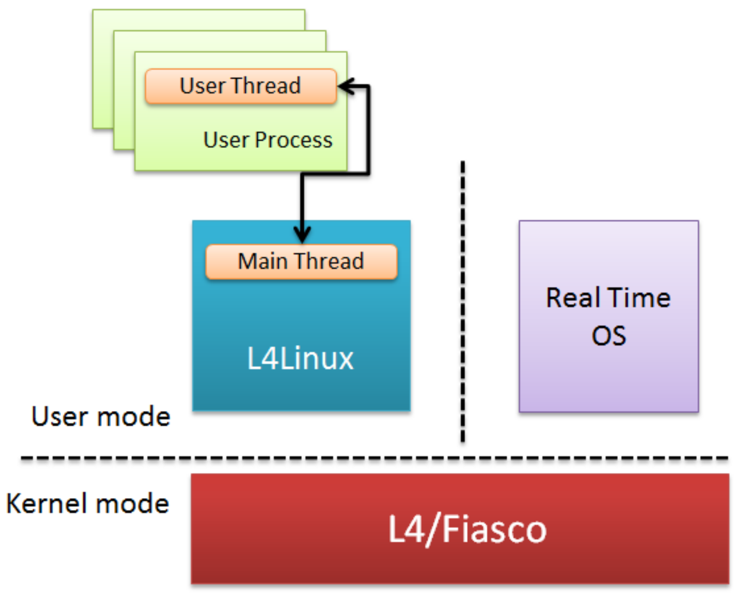


*Figure 4. General design of L4Linux*

By having L4Linux ported to L4/Fiaso, developers are spoiled with the abundant legacy applications of Linux. Real-time OS can be used to video or audio applications. By running it in different OS and on top of L4/Fiasco, it is assured that the application operations will not be interrupted.

# III. RELATED WORKS

## A. MPICH on Fiasco.OC

MPI standard which is used is MPICH has been tried to be ported and run on top of Fiasco.OC/L4Re. Fiasco.OC is a microkernel of the L4 family developed at the OS chair of TU Dresden. The L4Re refers to The L4 Runtime environment which constitutes the most basic part of its user-land. Michael Jahn's works are to show that MPI applications can be compiled on top of Fiasco.OC/L4Re and port a simple process manager to launch the MPI processes.

The contribution of this work lies in the porting the MPI library to Fiasco.OC. The spanwer library to spawn a process and endpoint library to maintain a channel were ported to support the gforker process manager to imitate the fork( ) process.

Several benchmark tests have been done by the author for point-to-point operation and collective operation by using several applications such as SkaMPI and Perftest. By considering many limitation existing in his development, most of the functionalities of MPICH can run properly over Fiasco.OC/L4Re.

## B. The PARAS Mikcrokernel

Parallel Machine 9000 (PARAM 9000) was a supercomputer at the Center for Development of Advanced Computing in Bangalore, India. The hardware is allowed to configure nodes as either compute or service nodes. While compute nodes were dedicated to execute workloads such as MPI exclusively, service nodes additionally provided filesystems and networking. All service nodes were operated with Sun Solaris. Compute nodes had two possible OS personalities. Sun Solaris was used in the cluster personality, where compute nodes were connected by LAN and provided a feature rich environment to users. PARAS was used in the Massively Parallel Processing (MPP) personality. This personality offered a highly optimized communication subsystem as well as Parallel Virtual Machine (PVM) and MPI for parallel programming. PARAS implemented threads and task creation, regions (virtual memory), ports, and messages inside the kernel. On top of that system services to provide names to ports, filesystems, and process control were implemented. Additionally a partition manager was running on any of the compute nodes to maintain information and manage resources for all nodes in the partition. The service partition, consisting of service nodes, was responsible for placing tasks on the compute partition and service I/O request.

## C. IBM Blue Gene – The Compute Node Kernel (CNK)

Blue Gene is a modular supercomputing platform that can scale up to 20 petaflops2. Common to all Blue Gene systems is the architecture that subdivides nodes by function. Similar to the PARAM 9000 approach, there are compute nodes and I/O nodes. Both node types can be grouped into variably sized sets of nodes, also known as blocks, partitions or partition sets (psets). The hardware ensures electric isolation of the communication network among the configurable blocks. The Blue Gene series uses a torus network for peer-to-peer communication, a collective network, and an interrupt network for fast barriers [BKM+10, Ch. I]. While I/O nodes run Linux, compute nodes are operated by a custom operating system kernel, the Compute Node Kernel (CNK). This kernel is extremely simple (5000 lines of code) and cannot run more than one process.

# IV. MPI on L4LINUX OVER FIASCO.OC MICROKERNEL

The goal of porting MPI library in L4linux over L4/Fiasco.OC microkernel is to show that the MPI-based application which is made by using parallel programming concept should perform better to overcome challenge in many-core systems. Therefore, the main task is to build the MPI library on L4/Fiasco as well as port it to L4Linux and design a scenario to test whether this parallel programming can optimize the performance of many-core systems on Microkernel Operating System.

## A. Design

There are two design scenarios to port the MPI library on L4/Fiasco, by porting the library to L4Re and by porting library to L4linux. These two scenarios are different in term of application. The library as well as MPI-based application is free to be implemented in L4Re directly. However, the implementation in L4Re is different with Operating System term. Even though, we do not see this because L4Re provides the interface that is similar to Linux (e.g., a POSIX C library, a virtual file system, etc.). Therefore, we have to adapt any kind of load balancing, process distribution, etc. to use the respective Fiasco and L4Re services. Therefore we have the second scenario to port the MPI library over L4Linux as virtualization. The concept of the second scenario is to run the application over the L4Linux, which this L4Linux is also a kind of L4 application. Those, the system call from the applications are not sent to the Fiasco Kernel but rather to the L4Linux kernel. The combination of the L4Linux kernel and our MPI-based application running on top of it is what we call a virtual machine (VM). We believe the virtualization may make sure that everything works as expected. This term can be illustrated with figure 5.

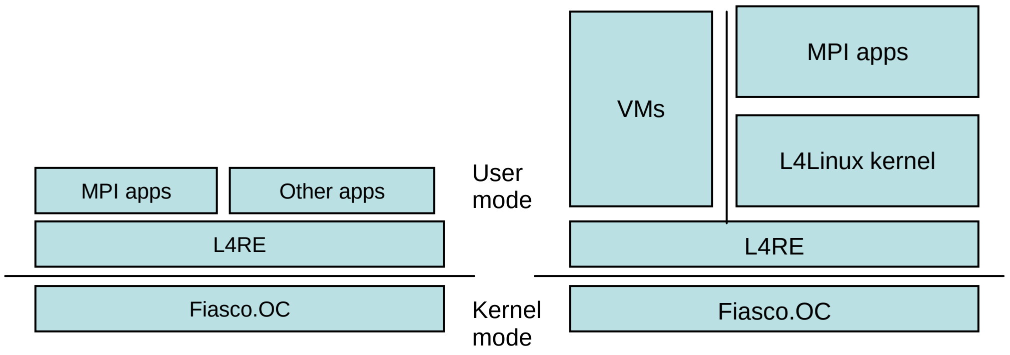


***Figure 5. Design porting MPI library on 2 different scenarios***

For the first scenario, we need to port the MPI library and compile the application on L4Re directly. There are two components of L4Re that should be designed; which are the user-level libraries and initial task loader, which is shown in figure 6. The design of porting the MPI library follows the instruction provided by Jahn. There are three main libraries that should be ported to L4Re; which those libraries are MPICH library, endpoint library, and spawner library. The last part that should be ported is the process manager, which has function to initialize the program, by linking the existing libraries that has been ported before. Then this process manager later can be invoke by initial task loader.

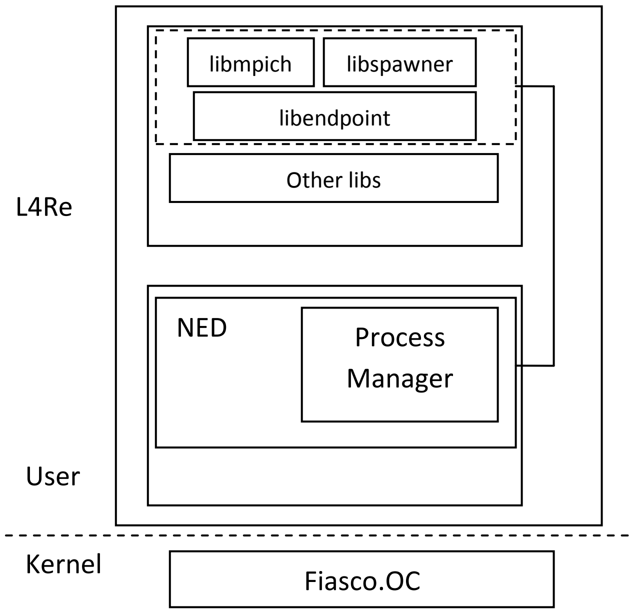


***Figure 6. L4Re components for porting MPI Library***

For the second scenario, we need to port MPI library and compile the MPI-based application on top of L4Linux. The virtualization over L4/fiasco will make sure everything to be done as expected. To port the MPI library on top of L4Linux is to build the library and compile the application on Linux kernel. After that put the related library and compiled application inside bootable L4Linux ramdisk.

## B. Implementation

### 1. L4Re

There are three steps of launching the MPI-based program; see figure 7. The first one is creating and launching the process. This is done by the process manager which acts as loader. In order to imitate the fork( )/execve( ) mechanism for process creation and binary execution, the libspawner is needed. The second step is installing control channel for MPI application. Libendpoint can handle this step by maintaining the pipes for process creation and messages labelling. And after those two steps are established, then each process should calls the MPI_Init( ) function of libmpich to begin the initialization. The other libraries are also implemented to support the application. Then to port any library to L4Re, we can see figure 8.

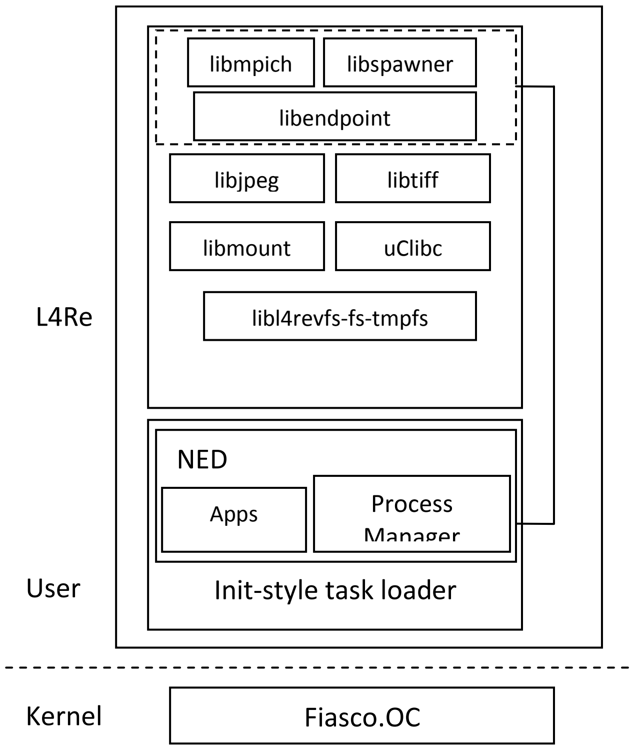


***Figure 7. The implementation of porting MPI library in L4Re***

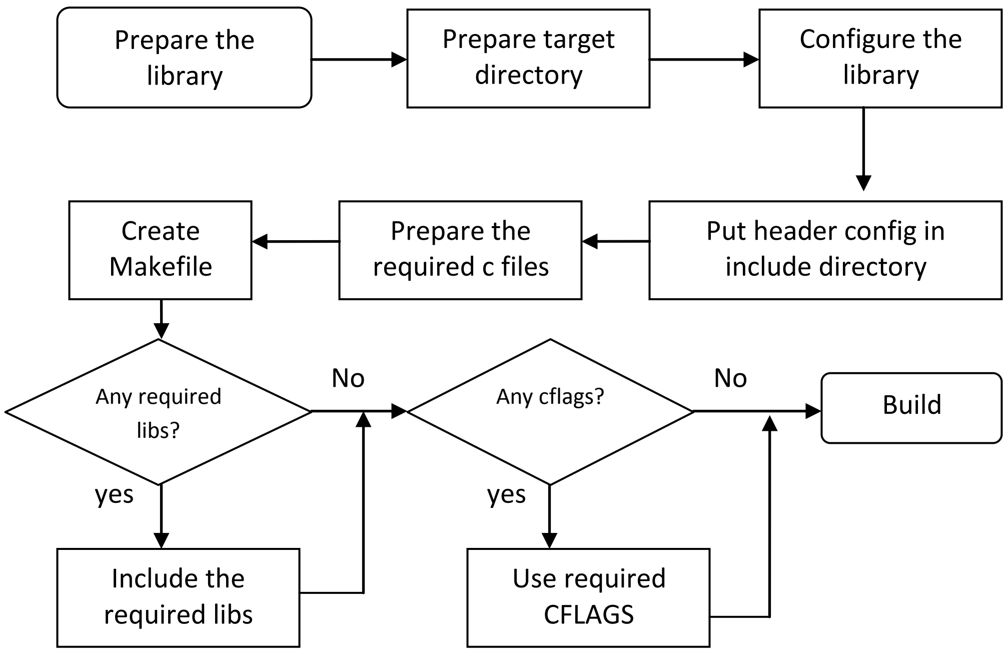


***Figure 8. Flowchart to port libraries to L4Re***

This procedure to port the library is also able to be applied in building any application. Hence, all of the library that is needed to be ported should follow this procedure. The compiler that is used by L4Re is uClibc which is different with the Linux compiler which use gcc. Therefore, some adjustments are needed to be able to compile the desired library in L4Re. The porting process will be explained for detail in appendix part.

## 2. L4Linux

The implementation of porting MPI library and the other libraries on L4linux should be done in Linux kernel. This should be done with the same architecture, such as x86 or AMD64. Compile the application that wants to be run in L4Linux with the related libraries. If the build and compilation process have been done, then the next section is store the necessary library and application to the L4Linux storage. Since the original storage has really small capacity that surely won’t fit to our library, then we have to create our custom storage that the capacity should be bigger than the current capacity. Once this process finished, the L4Linux is ready to be run.

The last part of implementation part is to make sure the kernel configuration in Fiasco.OC and L4Linux kernel. The configuration should match each other, because technically the L4Linux is also an application of L4Re which the L4Re need the Fiasco.OC to manage the hardware resources. Therefore the communication between the application and L4Linux kernel doesn’t mean the isolated communication without involving the L4/Fiasco microkernel..

# V. PERFORMANCE EVALUATION

## A. Simulation Environment

For the performance evaluation of the MPI on the microkernel, we are using QEMU x86 architectures. This QEMU can execute the guest code directly on the CPU. Therefore it can achieve near native performances. Since we are observing the behavior of many-core system, we use the AMD 32 cores system to run our experiment. The complete simulation parameters are summarized in table 1.

**Table 1. Simulation parameters for AMD 32 cores**

| **Description** | **Value** |
|---|---|
| Simulation Tool | QEMU 0.14 |
| L4Re Version | February 2014 release |
| L4Linux Kernel Version | Linux 3.7 |
| **Host** | |
| Machine | AMD Opteron |
| Host CPU Architecture | x86 64 Bit |
| Number of Cores | 32 |
| Memory | 132 GB |
| **Guest** | |

| | |
|---|---|
| Machine | X86 |
| CPU Architecture | i386 |
| Number of Cores | 32 |

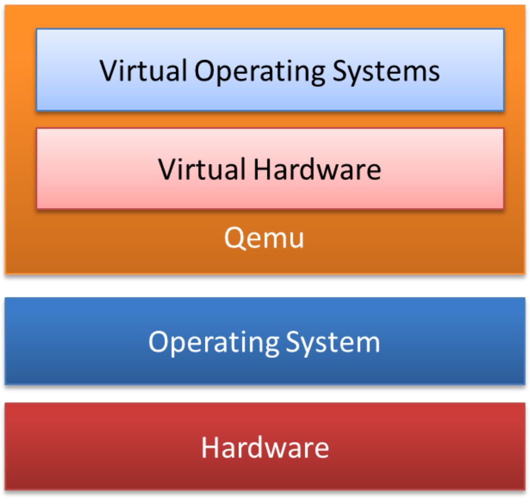


***Figure 9. Simulation environment using QEMU***

Figure 9 shows the relation between the host machine and guest machine. Those simulation environments are stated for L4Linux experiment. However, we have ported the MPI-library on L4Re. Therefore to check whether the ported version of MPI library in L4Re can run properly, we should test it. The simulation parameter follows the x86 architecture.

The L4Linux scenario can be classified into 3 kinds. The experiment of benchmark program will be done using MPBench benchmarking tool. This benchmark scenario is done to show the feasibility of MPI functions. The second scenario will be done using matrix multiplication application which is parallelized with MPI library. The third scenario will be done using image processing application. The detail of experiment scenario and parameters are shown in table 2.

**Table 2. Experiment scenarios and parameters**

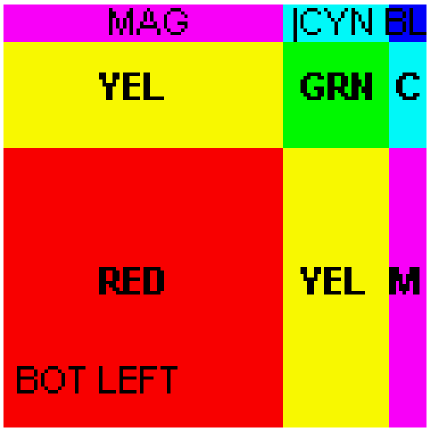
| Scenario | Parameter | Value |
|---|---|---|
| Scenario 1<br>Benchmark tool | Number of cores | 32 |
| | Number of processes | 32 |
| | Other MPBench parameters | Default |
| Scenario 2<br>Matrix multiplication | Number of cores | 32 |
| | Number of processes | 2, 4, 8, 12 |
| | Linux's RAM | 132 GB |
| | Linux architecture | 64 bit |
| | L4Linux RAM | 1.2 GB |
| | L4Linux architecture | 32 bit |
| Scenario 3<br>Image processing | Number of cores | 32 |
| | Number of processes | 1, 2, 4, 8 |
| | Linux's RAM | 132 GB |
| | Linux architecture | 64 bit |
| | L4Linux RAM | 1.2 GB |
| | L4Linux architecture | 32 bit |

## B. Benchmark and Test Programs

Our benchmarking program is MPBench. It is a program to measure the performance of some critical MPI functions. This benchmarking tool run the test to check several functions, such as send/receive function, broadcast function, reduce function, and all-to-all function. The first test program is about matrix multiplication. The second program is about image processing program which is called Sobel-step program. It consists of wrapper to mpi communication and image processing program. Since our goal is to measure the performance of using mpi on many-core system, then the parameter comparison should be processing time. This complex program is supposed to be run on the L4Linux on top of L4/Fiasco microkernel.

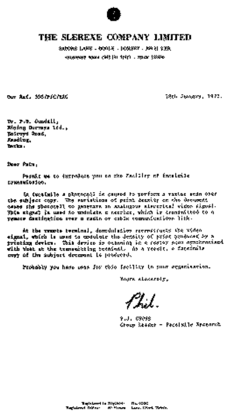
In image processing application test, we have several images to be observed. Those kinds of images can be seen in figure 10.

(a)



(b)

(c)



THE SLEREXE COMPANY LIMITED

[illegible]

[illegible]

Our Ref. 350/PJC/EAC 18th January, 1972.

Dr. P.N. Cundall,
Mining Surveys Ltd.,
Holroyd Road,
Reading,
Berks.

Dear Pete,

Permit me to introduce you to the facility of facsimile transmission.

In facsimile a photocell is caused to perform a raster scan over the subject copy. The variations of print density on the document cause the photocell to generate an analogous electrical video signal. This signal is used to modulate a carrier, which is transmitted to a remote destination over a radio or cable communications link.

At the remote terminal, demodulation reconstructs the video signal, which is used to modulate the density of print produced by a printing device. This device is scanning in a raster scan synchronised with that at the transmitting terminal. As a result, a facsimile copy of the subject document is produced.

Probably you have uses for this facility in your organisation.

Yours sincerely,

Phil.

P.J. CROSS
Group Leader - Facsimile Research

[illegible]

(d)

*Figure 10. Observation images for testing image processing application: (a) francois.tif, (b)flag.tif, (c)marbles.tif and (d) text.tif*

## C. Simulation Results and Analysis

### 1. MPBech

The first figure shows the MPI bandwidth in unidirectional MPI. Figure 11 shows the performance of L4Linux in running the MPI function. The bandwidth of the message passing in this experiment seems to be correct because it is keep rising as the increasing of the message size. Because of the implementation of MPI in single machine, the message size affects the performance. The bandwidth reaches the saturation point at 1MB of message size. While the bandwidth keep decreasing after reaching 33MB. Figure 12 shows broadcast function in MPI. This result is quite similar with the bandwidth result and the analysis is also quite similar as well.

Figure 13 shows the all-to-all MPI function. This kind of experiment test the feasibility of each process to sends to every other process. The graph shows that each process can complete send and receive to each other between 33kb until 40 MB. And the last, figure 14 shows the roundtrip message for each various message size. The number of transaction for each second seems to be decrease as the increasing of the message size. This graph shows that this MPI roundtrip function seems to be run correctly.

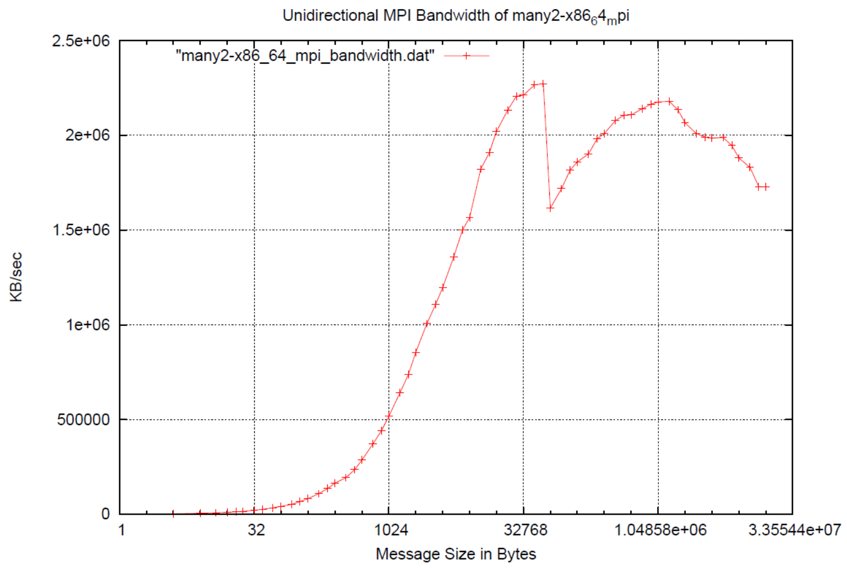


*Figure 11. Performance of MPI bandwidth*

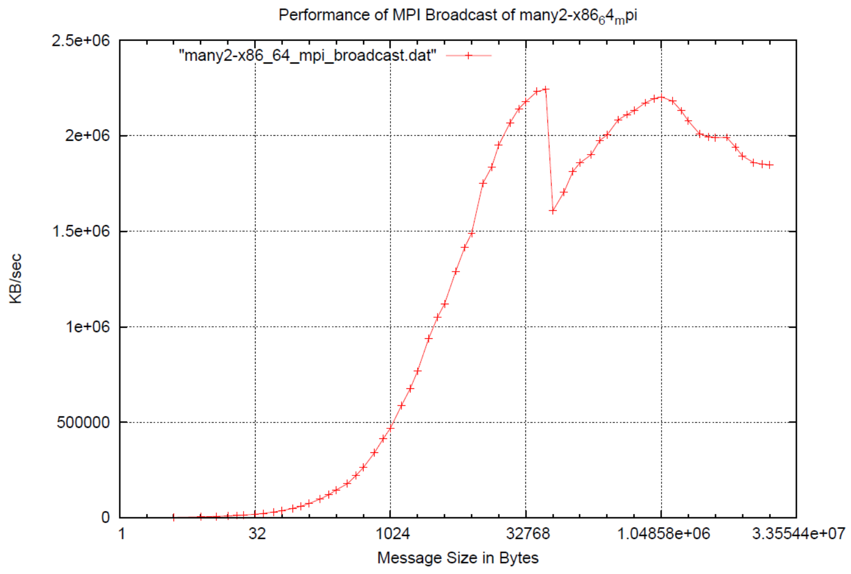


*Figure 12. Performance of MPI broadcast*

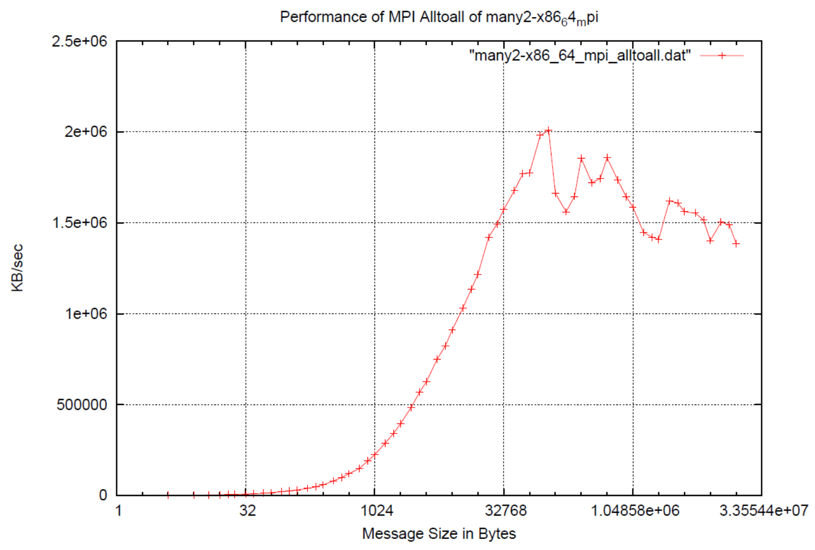


*Figure 13. Performance of MPI all-to-all*

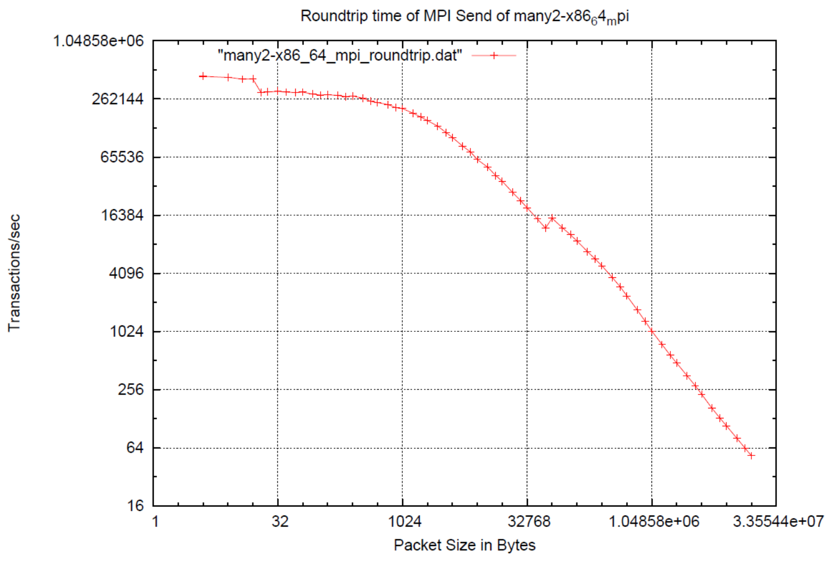


*Figure 14. Performance of MPI send-receive roundtrip*

## 2. Matrix Multiplication

Figure 15 shows the processing time of matrix multiplication application over Linux and L4/fiasco (L4linux). From the figure, we derive some information about how the number of processes takes effect to the processing time. In this experiment, the implementation of MPI in L4Linux seems to be better in the less number of processes. While the processing time in big number of process remains constant. In Linux, the performance seems to be steady and the speed up is always increase as the number of processes. Based on this experiment result, we can say that the consistency of MPI in big number of processes in L4/fiasco still can't be guaranteed. However, the improvement of using the MPI on L4/fiasco has been achieved. If we look the table 3, we can see the speed up over 2 processes. The speed up of this experiment is quite high. This is represented by the first speed up gain more than twice of the 2 process. So it means we achieve the speed up beyond our expectation.

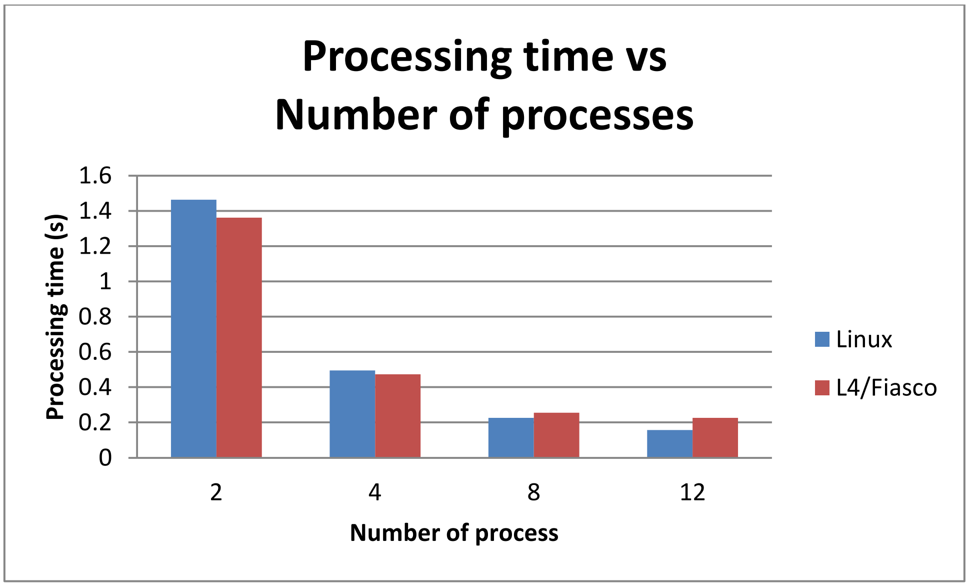


*Figure 15. Processing time of matrix multiplication application over Linux and L4Linux in various number of processes*

**Table 3. Speed up in various numbers of processes**

| Processes | Speed Up |
|---|---|
| 2 | 1 |
| 4 | 2.95 |
| 8 | 6.49 |
| 12 | 9.35 |

However, the improvement of large number of processes can't be shown in L4Linux result. If we compare it with the Linux result, the performance in Linux seems to be better. We believe this occurs because we run the application in different OS architecture which the Linux system uses 64 bit, while the L4Linux system uses 32 bit. As we know that the memory management in both architectures is slightly different. We can't use the large number of memory in 32 bit architecture. To get the fair result, the experiment should be done in the same architecture. However, the 64 bit L4Linux is not ready yet to be used. Therefore we use the 32 bit of L4Linux instead of 64 bit L4Linux.

In L4Linux, with the large number of processes, the cores that is used is also as many as the number of processes. Therefore, in large number of processes, each core will use a large amount of shared memory to process each process. Because of the amount of memory in L4linux is limited, we can't avoid the performance degradation in a large number of processes. However, our result still shows the better performance in term of parallelism on the microkernel system.

## 3. Image Processing Application

Figure 16 shows the processing time of image processing application over Linux with different images. We can see that all of the images experience the performance degradation as the increasing number of processes. This result is contradicting with our concept that by increasing the number of processes, the processing time should be decrease or there is an improvement in the performance. Before we discuss about this anomaly, we need to analyze the performance in L4Linux. Generally, the performance is quite the same in term of degradation of performance. But there is something important in figure 18, that the computation of the francois.tif image can only be measure until 4 processes. It happens because the structure of image is quite complex, so it affects the amount of computation and processes. The memory can't handle the amount of processes in processing francois.tif. It means the experiment is memory dependent. It seems to be as our expectation that in complex computation, the single machine could be met a complication in doing that task with a limited memory in single machine.

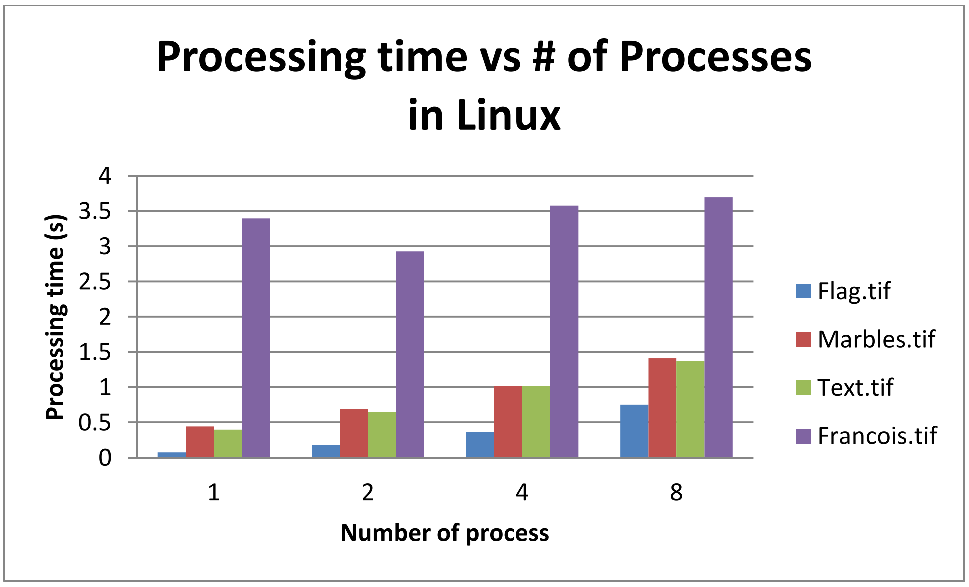


*Figure 16. Processing time of image processing application over Linux with different images*

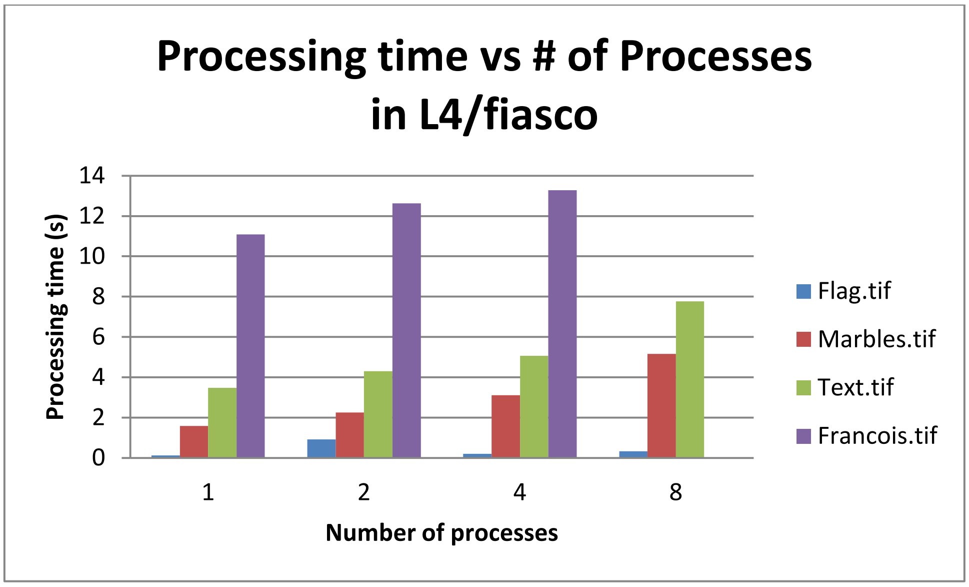


*Figure 17. Processing time of image processing application over Linux with different images*

However, the performance degradation should not be caused by the limitation of memory in single machine. Then, why in this experiment, the increasing number of processes can't guarantee the improvement of performance? There are several assumptions about this anomaly. First we have to look inside of the application. The image application that we used is consisting of hybrid implementation which is dominated by OpenMP implementation. OpenMP is kind of thread parallel programming which is using shared memory. Implementing OpenMP itself has improved the performance of the application (compared to the serial programming). However, the addition of the MPI as the message passing gave the system a harder workload as the increasing of message overhead. So, by the amount of overhead of MPI message, the increasing number of processes will not give any improvement for this scenario. Eventually, the image processing application is not suitable for this kind of experiment scenario.

# VI. CONCLUSIONS

The MPI library has been ported to measure or test the feasibility of MPI on L4linux over L4/fiasco microkernel for many-cores system. Several applications have been tested to show the feasibility of MPI on L4/Fiasco. The MPBench benchmarking experiment result shows that the functionality of MPI library was running properly. The improvement of performance is shown by the matrix multiplication application. As the increasing number of processes, the processing time is heavily decreased. The speed up of the performance also meets beyond our expectation that for each multiple of 2, the speed up gain more than twice or almost three times.

Unfortunately, the experiment of image processing could not meet our expectation that by increasing the number of processes, the performance becomes worse. However, this degradation of performance should not affect our conclusion that the MPI improves the performance of the many-core system. We agree that the experiment of image processing is not suitable for MPI on a single machine environment.

In conclusion, we can say that the MPI is feasible to be performed on top of L4Linux on L4/Fiasco microkernel for many-core system.

# BIBLIOGRAPHY

[1] Beleg, Großer. "MPICH on Fiasco. OC/L4Re." *Student project on Technische Universitat Dresden*. (2013).

[2] Gropp, William, Ewing Lusk, and Anthony Skjellum. Using MPI: portable parallel programming with the message-passing interface. Vol. 1. *MIT press*, 1999.

[3] Wilkinson, Barry, and Michael Allen. Parallel programming. Vol. 999. New Jersey: Prentice hall, 1999.

[4] Barney, Blaise. "Introduction to parallel computing." *Lawrence Livermore National Laboratory* 6.13 (2010): 10.

[5] Barney, Blaise. "Message passing interface (mpi)." *Lawrence Livermore National Laboratory, https://computing. llnl. gov/tutorials/mpi/,* available online 2010 (2009).

[6] Almási, George, et al. "MPI on BlueGene/L: Designing an efficient general purpose messaging solution for a large cellular system." *Recent Advances in Parallel Virtual Machine and Message Passing Interface. Springer Berlin Heidelberg*, 2003. 352-361.

[7] Buyya, R. "The design of paras microkernel." *Relatório técnico* (1995).

[8] Dresden, T. U. "The Fiasco microkernel." (2011).

[9] Schneider, Sven. Multiprocessor Support for the Fiasco Microkernel. Diss. Chemnitz, Germany: Technical University of Chemnitz, 2006.

[10] Partheymuller, Markus, Julian Stecklina, and Bjorn Dobel. "Fiasco. OC on the SCC." Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium. No. 55. Universitätsverlag Potsdam, 2012.

[11] Balaji, Pavan, et al. "MPI on a Million Processors." *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. *Springer Berlin Heidelberg*, 2009. 20-30.

[12] Mallón, Damián A., et al. "Performance evaluation of MPI, UPC and OpenMP on multicore architectures." *Recent Advances in Parallel Virtual Machine and Message Passing Interface. Springer Berlin Heidelberg*, 2009. 174-184.

[13] Clauss, Carsten, et al. "Evaluation and improvements of programming models for the Intel SCC many-core processor." *High Performance Computing and Simulation (HPCS)*, 2011 *International Conference on. IEEE*, 2011.

[14] Snir, Marc, ed. MPI--the Complete Reference: The MPI core. Vol. 1. *MIT press*, 1998.

[15] Rabenseifner, Rolf, Georg Hager, and Gabriele Jost. "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes." Parallel, Distributed and Network-based Processing, 2009 17th *Euromicro International Conference on. IEEE*, 2009.

[16] Warg, Alexander, Hermann Härtig, and Michael Hohmuth. "Software Structure and Portability of the Fiasco Microkernel." *Dresden: Technical University of Dresden* (2003).

[17] Beleg, Großer. "Exploring Inter-Core Message-Passing for Fiasco on the SCC." (2011).

[18] Diaz, Javier, Camelia Munoz-Caro, and Alfonso Nino. "A survey of parallel programming models and tools in the multi and many-core era." *Parallel and Distributed Systems, IEEE Transactions on* 23.8 (2012): 1369-1386.

[19] Hwu, Wen-mei, et al. "Implicitly parallel programming models for thousand-core microprocessors." *Design Automation Conference, 2007. DAC'07. 44th ACM/IEEE. IEEE*, 2007.

*[20]* Perumalla, Kalyan S. "μsik-a micro-kernel for parallel/distributed simulation systems." Principles of Advanced and Distributed Simulation. *Workshop on IEEE, 2005.*

# ACKNOWLEDGEMENTS

First and foremost, I express my sincerest gratitude to my supervisor Prof. Moonsoo Kang for his guidance, caring, patience that also providing me with comfortable research environment. His sincere encouragements to pursue my master's degree and his continuous support have been a great assistance to achieve a milestone in my career. Without his help, this thesis would not have been completed. He is the friendliest, easy going and encouraging advisor that anyone could wish for.

I would also like to show appreciation to all members of the thesis examining committee Prof. Jaehong Shim and Prof. Sangman Moh for their valuable advices and insight throughout my research. In addition, I would like to thank Department of Computer Engineering, Chosun University, to provide me the atmosphere to augment my knowledge.

My family members in Indonesia have always supported me with their care and love from far away. The experience of being away from home helps me to realize how important and valuable my family is. Therefore, I am grateful to have them and I would like to dedicate this thesis to my family.

During my daily life in Korea, I have been supported by colleagues and cheerful friends. Ganis Zulfa Santoso is my lab mate as well as my senior who always guide me in my research. I'm really grateful to him for giving me so much care during my research to complete this thesis. Without his help, this thesis will not complete successfully. Donggeun Cha is my Korean friend who always accompanies me everywhere I go. With his help, I could survive to live in Korea. During my thesis defense presentation, I was help so much by Shelly Salim which she helped me prepare my thesis defense from the beginning. I express my gratitude for her assistance during my thesis defense. My fellow-countrymen Adrianto Tejokusumo, Rico Hartono, Christian Oey, Ivan Christian, Yvan Christian, Perdana, Ardiansyah, Muhammad Nugraha, and other Indonesian friends in Gwangju, who I can’t mention it one by one, have made my life in Korea more interesting and convenient.

This research was supported by the Global IT, under the NIIED (National Institute for International Education).