INF-3981

MASTER'S THESIS IN

COMPUTER SCIENCE

Efficient Intra-node Communication for Chip Multiprocessors

Torje S. Henriksen

October, 2008

FACULTY OF SCIENCE
Department of Computer Science
University of Tromsø

INF-3981

MASTER'S THESIS IN

COMPUTER SCIENCE

Efficient Intra-node Communication for Chip Multiprocessors

Torje S. Henriksen

October, 2008

Abstract

The microprocessor industry has reached limitations of sequential processing power due to power-efficiency and heat problems. With the integrated-circuit technology moving forward, chip-multithreading has become the trend, increasing parallel processing power. The shift of focus has resulted in the vast majority of supercomputers having chip-multiprocessors.

While the high performance computing community has long written parallel applications using libraries such as MPI, the performance-characteristics have changed from the traditional uni-core cluster, to the current generation of multi-core clusters; more communication is between processes on the same node, and processes run on cores sharing hardware resources, such as cache and memory bus.

We explore the possibilities of optimizing a widely used MPI implementation, Open MPI, to minimize communication overhead for communication between processes running on a single node. We take three approaches for optimization: First we measure the message-passing latency between the different cores, and reduce latency for large messages by keeping the sender and receiver synchronized. Second, we increase scalability by using two new queue-designs, reducing the number of communication queues that need to be polled to receive messages. Third, we experiment with mapping a parallel application to different cores, using only a single node. The mapping is done dynamically during runtime, with no prior knowledge of the application's communication pattern.

Our results show that for large messages sent between cores sharing cache, message-passing latency can be significantly reduced. Results from running the NAS Parallel Benchmarks using the new queue-designs show that Open MPI can increase its scalability when running more than 64 processes on a single node. Our dynamic mapper performs close to our manual mapping, but rarely increases performance.

We see from the experimental results, that the three techniques give performance increase in different scenarios. Combining techniques like these with other techniques, can be a key to unlocking the parallel performance for a broader range of parallel applications.

Acknowledgments

I would like to thank my supervisors Otto J. Anshus, Lars Ailo Bongo and Phuong Hoai Ha for the massive support I have received during the work with this thesis. Special thanks goes to Otto for helping me find a subject for my thesis and introducing me to Phuong and Lars for whom I have endless respect and have enjoyed working with.

I also want to thank the people at the Department of Computer Science. In particular Jan Fuglesteg for help with all kinds of formalities, Maria W. Hauglann for help getting access to the Stallo cluster and Jon Ivar Kristiansen for help with the smaller cluster at the university. Thanks to the HPC group at UiT for donating CPU time at Stallo, and in particular Steinar Henden for technical support.

The members of the Open MPI mailing list have helped me understand the inner workings of Open MPI. My fellow students and friends have helped me tremendously with comments and discussion, and most of all encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

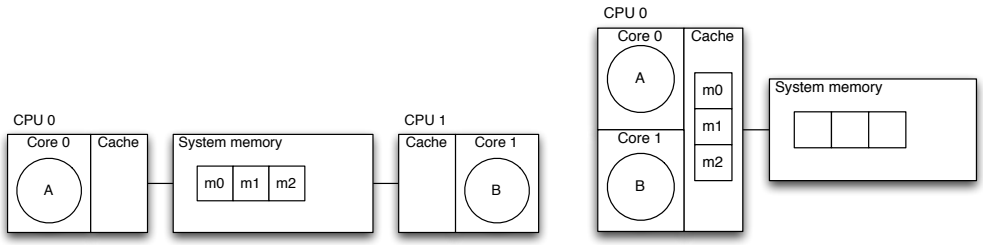
Computational power that can currently only be delivered by supercomputers, is needed in a wide specter of scientific areas, including environmental, financial and medical research. However, the supercomputers available today are not fulfilling their full potential, and a 70% processor utilization is considered very good [28]. The processors are limited by the relatively smaller improvements in memory access latency and communication latency over the last decade [1].

As the microprocessor industry has reached limitations in sequential performance due to power consumption and heat, the advances in integrated circuit technology is now used to increase parallelism for processors [4]. This has resulted in chip-multithreading (CMT) technologies such as simultaneous multithreading (SMT) and chip-multiprocessors (CMP).

According to the Top500 list (June 2008) [34], 80% of the supercomputers are now clusters of workstations with CMP-technology. The typical cluster node has two CPU-sockets, where each CPU has two or four cores. The nodes are connected with Ethernet or high-speed interconnects such as Infiniband.

In addition to the general purpose CPU, we also have other highly parallel processors, such as graphical processing units and the Cell Broadband Engine. The Cell is used along with general purpose Opteron processors to create the world's fastest supercomputer, the Roadrunner, performing at 1PFlops. GPUs also get attention from the scientific community, and both nVidia and AMD/ATi are developing libraries and languages to allow general purpose computing on GPUs (GPGPU or GPGP). Grid projects such as Folding@Home [9] also explore the possibilities of Cell and GPU by having clients specifically developed for those platforms.

According to [4], up to 50% of communication in general purpose of clusters can be intra-node communication, referring to communication between processes sharing memory. This indicates that improving performance of



(a) Not sharing cache (b) Sharing cache

Figure 1.1: Intra-node communication. Two processes A and B, running at different cores communicating through a shared memory queue. In a) A and B do not share cache, while in b) A and B share cache.

intra-node communication can reduce overall execution time, and even more so as the number of cores per node increases.

Intra-node communication is most often based on shared memory queues, where the sending and receiving process sends and receives messages by accessing a queue residing in shared memory. In some hardware architectures, such as the current generation of Intel Xeon processors, pairs of processor cores on a node share not only memory, but also cache. Others share only memory. Figure 1.1 illustrates a shared memory queue, where in a) the queue is accessed through the memory bus, while in b) the processes share cache and do not have to go through the memory bus. Shared cache is one example of how the communication-path between cores can vary even on the same node. Other shared resources such as the memory bus, can also affect the performance of communication.

It has earlier been suggested that both operating systems and middleware should be multi-core aware [4, 8], to better utilize shared resources such as cache and memory bus. Work has been done to increase scalability in multi-core systems, and a topology aware message passing protocol for intra-node communication has also been suggested [6].

The Message Passing Interface (MPI) is a widely used standard for message passing middleware. Common MPI implementations such as Open MPI [26] and MPICH2 [21] support and differentiate between communication over different interconnects such as Infiniband, Ethernet and shared memory, but optimization for intra-node communication for specific hardware architectures is lacking.

We propose three approaches in order to optimize intra-node communication in CMP clusters. First we explore the performance of the different communication-paths within a node. We then demonstrate a significant

performance increase for large messages sent between processes sharing L2-cache. Second, the scalability of the message-passing queues is explored and improved for many applications. Last the mapping of processes to processor cores is explored, mapping communicating processes closer to each other, using only a single node. The mapping is first done statically by manually analyzing the application. Then we design and implement a dynamic mapper, mapping applications during runtime, based on communication patterns logged by MPI. To our knowledge, the possibilities for such a mapper, has not earlier been investigated on the current generation of supercomputers.

Message passing latency is measured using a micro-benchmark, testing all different communication-paths for intra-node communication. The NAS Parallel Benchmark-suite [22] is used to benchmark the scalability of the queues, by running up to 128 processes on a single eight core node. For static mapping, we do a case study on the CG NAS Parallel Benchmark, and use the CG, SP and BT for the dynamic mapper.

We find that performance of message-passing is highest between processes sharing cache, but that care has to be taken to utilize the cache for large messages. This is done by keeping the sender and receiver synchronized, and significantly improves performance for large messages. The scalability of the message-passing queues is significantly improved using two queue-designs that limit the number of queues the receiver needs to poll when the number of processes per node becomes high. We were not able to improve performance of any of the NAS Parallel Benchmarks using static or dynamic mapping, except for very few cases where we run more than eight processes on a single node. In these specific cases, performance was improved by up to 10%, and the dynamic mapper performed comparable to the static mapper.

Our experimental results show that we have improved performance for some scenarios; when running more than 64 processes on a single node, sending large messages between processes sharing L2-cache, or when running applications behaving like the CG benchmark, being overdecomposed. To minimize communication overhead in these situations, these techniques can all be used. However, to unlock the parallel performance of today's CMP supercomputers, more techniques are needed. Component based communication middleware such as Open MPI can be a powerful tool to combine optimizations, and if necessary apply the optimizations beneficial for the current hardware and software at execution or even runtime.

The rest of the thesis is organized as follows. Chapter 2 gives a background overview of topics important to intra-node communication and mapping in chip multithreaded systems. Chapter 4 describes the architecture of the hardware we run experiments on. The architecture of Open MPI is also described, and components important for intra-node communication are de-

scribed in detail. In chapter 5 we explore the performance of message passing latency in our research platform, and propose and implement improvements for large messages sent between cores sharing cache. Chapter 6 describes two queue-designs that improve scalability compared to the Open MPI standard queues. The queues are then evaluated experimentally using the NAS Parallel Benchmarks. In chapter 7 we present the design of a dynamic mapper, as well as experiments demonstrating how mapping of processes on a single node can impact performance. We conclude the thesis and outline future work in chapter 8.

Chapter 2

Background

To help understanding the work presented in the thesis, we will in this chapter give an overview of the mechanics of intra-node communications and describe the hardware of today's CMT-systems found in workstations and high performance computers. A short introduction to theory behind mapping is also given.

2.1 Intra-node communication

By intra-node communication we refer to communication between threads running on the same node. The node may have one or several CPUs, and the CPUs may have one or more cores.

The communication can further be divided into communication between threads running in the same process (sharing address space), or inter-process communication (IPC). Communication between threads sharing address space is an exercise in synchronizing the threads, not letting them write to the same data at the same time. Communication between processes not sharing address space requires data to be copied from one address space to another. Splitting an application into several processes with their own address space makes it easier to spread the application to several nodes, and is used by the MPI-implementations. We will only consider inter-process communication as we will work with MPI.

Chai et al. [5] classifies the different mechanisms of inter-process, intra-node communication into three categories that we discuss briefly to get an overview of the topic. The three proposed categories are:

- NIC-based loopback.
- Kernel-assisted memory mapping.

- User space memory copy.

The following three sections give an overview of these categories, and discuss how they perform and their portability across platforms.

2.1.1 NIC-based loopback

The primitives used by inter-node communication such as TCP and UDP sockets, can also be used by processes located at the same node. It is common that the operating system provides a loopback interface, doing the message passing without needing a physical network interface card (NIC). If a NIC is available it can detect if the sender and receiver reside at the same node (shared IP-address) and loop-back the message. By loop-backing, the latency of the network is avoided as the message is never put on the network link.

An advantage of using NIC-based message passing is that there is no need to distinguish between intra-node and inter-node communication, making the implementation of user-applications and middleware simpler. However, projects that focus on high performance computing will distinguish between them to optimize intra-node communication where possible.

2.1.2 Kernel-assisted memory mapping

Kernel-assisted memory mapping utilizes features provided by the operating system kernel to copy data from the address space of one process, to the address space of another process. The kernel needs to be involved, as the processes each have their own private address space out of limit of the other processes. The advantage of doing a direct copy from one process to another, is that it only takes one copy, compared to two copies needed when using shared memory mapped in user space (described later). This reduces the cache-pollution and can possibly reduce the latency of message passes. There are different ways of doing kernel-assisted memory mapping, depending on the operating system and version. Two possible approaches used in Linux are Ptrace and Kaput that are both explored in [2].

Kaput lets a process register some of its memory and map it into kernel space. Processes can communicate by doing *put* and *get* on this area, and can copy directly to the address space of another process, as long as it is registered by Kaput.

Ptrace is a system-call that provides the means to control one process from another. The process in control is called the parent, while the controlled process is called the child. This does however not mean that the parent has to create the child. The Ptrace system-call is primarily used for writing

debuggers and tracking system calls, but also makes it possible for one process to read and modify the memory of another process on the same node. This can be used to copy data from one process' address space directly to another process' address space.

There are however a few drawbacks. First, Ptrace is a system call which increases latency as the kernel has to be involved. Another drawback is that the process being controlled is frozen while the parent is controlling it, making it harder to overlap communication with computation. A third drawback is that the sender and receiver have to be synchronized. As the sender will put the message directly into the address space of the receiver, the receiver must have a receiving buffer ready, and also provide the sender with the address of the buffer. This means that the sender needs to wait for the receiver to do a receive call. It cannot just put the message in a shared buffer and expect the receiver to fetch it when wanted.

2.1.3 User space memory copy

User space memory copy is a technique where the processes share some memory region that is used as a buffer or queue. When a process wants to send a message, the message is copied into this shared memory. When the receiver process is ready to receive the message, it can fetch it from this shared memory. If the sender has not sent the data when the receiver needs it, the receiver has to wait.

The advantages of this technique are that the sending and receiving do not require any system calls, and it is more portable across platforms than the kernel-assisted memory mapping. It does however require primitives for doing shared memory, and this can be done differently across operating systems. The drawback is mainly the use of two copies, compared to one copy when using kernel assisted memory mapping. The extra copy adds to the latency and pollutes the cache by accessing the shared memory region for both sending and receiving a single message.

Table 2.1 summarizes what methods require system calls for message passing. The NIC-based loop-back can support one-sided copying reducing the number of copies needed, but this is hardware dependent.

Now that we have covered some of the possible ways of doing inter-process, intra-node communication, we will move on to describing the design of chip multiprocessors found in most of today's high performance computers.

Table 2.1: System calls and number of copies

Communication-mechanics	System calls for message pass	Number of copies per message pass
NIC-based loop-back	yes	2
User space memory copy	no	2
Kaput	yes	1
Ptrace	yes	1

2.2 Chip multithreading

There are five abbreviations that are frequently encountered when reading about chip multithreading. CMT is one of these, and is explained immediately along with the other four.

- SMT - simultaneous multithreading
- CMP - chip multiprocessor
- CMT - chip multithreading
- SMP - symmetric multiprocessor
- NUMA - Non-unified memory access

While superscalar processors allow multiple instructions from the same thread to run in parallel, SMT allows for multiple instructions from multiple threads to run in parallel. This can increase processor utilization when facing memory latencies and limited parallelism per thread. SMT [35] should not be confused with multicore, where each core can execute threads in parallel. The Intel Pentium 4 [13] and SUN's Niagra [17] are examples of processors that support SMT.

CMP [25] refers to implementing two or more processor-cores on the same chip. Chip multiprocessors are also named multi-core processors. Each core in an CMP can be designed to support SMT.

CMT is an expression that includes both SMT and CMP, referring to processor-chips that support more than one simultaneous hardware thread using either SMT, CMP or both.

Symmetric multiprocessor (SMP) refers to systems where two or more processor cores reside on the same node, either at the same CPU (multi-core) or at different CPUs.

NUMA is a technology developed to increase scalability of SMP systems. Memory is shared between processors, but the memory access time depends on the memory location relative to a processor. Keeping the data in memory close to the process using it, is crucial for efficient use of NUMA systems.

To understand the performance of CMP systems, it is important to know how they are designed. We will now describe the evolution of CMPs, so that we can better understand the design of today's CMPs and how it effects performance. SMT and NUMA systems will not be covered in detail in this thesis.

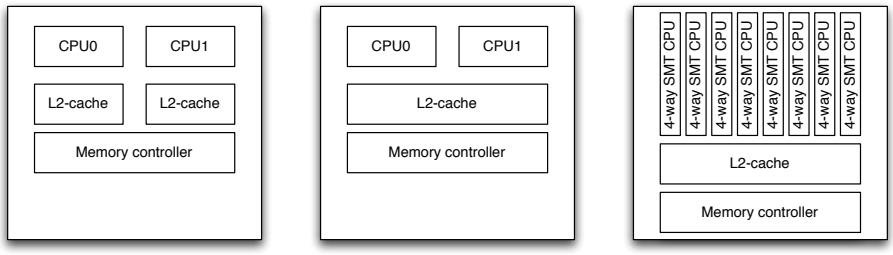
2.2.1 The evolution of CMP

Spracklen et al. [33] describe the evolution of chip multithreading in the industry, using SUN's [23] offerings as the representatives. The evolution is divided into three generations, illustrated in figure 2.1.

The first generation is represented by *Gemini* and *Jaguar* dual-core derivatives from the UltraSPARC-II [16] and UltraSPARC-III respectively. They are identified by having no shared cache or other links between the cores. They are practically two processors built on the same chip, sharing a memory bus.

The second generation is represented by the UltraSPARK-IV+ [14], where the on-chip L2-cache and off-chip L3-cache were shared between the two cores. A shared L2-cache allows for faster communication between running threads, and also keeps the cache coherent. When the processor is used for a just a single-threaded application, it has access to the whole cache.

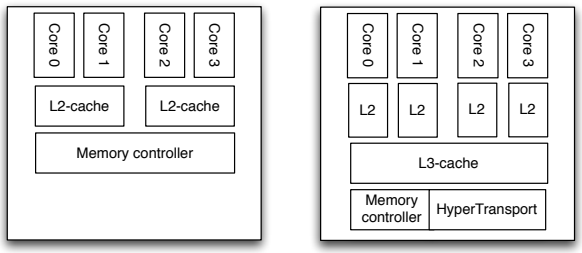
The third generation is represented by *Niagra*. The entire design of Niagra, including the design of the cores, is optimized for a CMT design point.



(a) 1st generation (b) 2nd generation (c) 3rd generation

Figure 2.1: CMT evolution. Figure is based on a figure in [33]

To put today's quad core CMP offerings from Intel and AMD into the context, Intel on one side is still at the second generation. Even the quad core CPUs are built of two second generation dual-cores. AMD on the other hand builds quad cores where the L2-cache is private for each core, while an L3-cache is shared between all four cores. AMD refers to their quad core to *true* quad core technology as the entire design is centered around CMT, not derived from previous uni-core designs. The Intel and AMD offerings are illustrated in figure 2.2.



(a) Intel Clowertown (b) AMD Opteron

Figure 2.2: Current quad CPU offerings from AMD and Intel

Another difference between the Opteron 2.2 b) and Xeon in 2.2 a) is that the Opteron utilizes HyperTransport to scale bandwidth to memory and between cores. While HyperTransport is an open standard, Intel has developed their own technology named Common System Interface, often referred to as Quickpath. This technology will be available on the next generation of Intel CPUs.

2.3 Mapping

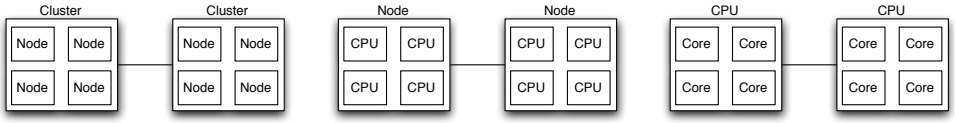
Mapping refers to the task of assigning processes to processing units. The environment can be grid, clusters of workstations, massively multiple processors or any parallel architecture. The applications have two attributes we can base the mapping on:

- Communication-pattern
- Processing demands

Communication-pattern further has two attributes; number of messages and size of messages. Processing demand refers to the amount of processing power needed to finish the process. When mapping by processing demands, the goal

is to map processes to minimize *turnaround time*; the execution time of the longest running process. Processes needing more processing power can be scheduled to faster CPUs in heterogenous systems, or processes demanding much memory can be scheduled across nodes to utilize more of the memory in the system.

When mapping is based on communication pattern, the goal is to minimize communication overhead. This can be done by mapping communicating processes *closer* to each other. What is considered close depends on the hardware architecture, but close generally means having a fast communication channel (interconnect). In a grid environment where several clusters are connected through the Internet, communication overhead can be minimized by having as much as possible of the communication being intra-cluster (LAN), as it is in general faster than inter-cluster communication (Internet). This concept can also be used in multi-core clusters. Communicating processes can be mapped close to each other by letting them run on the same node, utilizing shared memory communication. Furthermore the concept applies for intra-node communication, where it is possible that some cores are closer to each other than others. For example two cores can share cache, providing fast communication. Figure 2.3 illustrates these three scenarios where mapping can be applied to minimize communication overhead.



(a) Grid (b) Cluster (c) Node

Figure 2.3: Three scenarios where mapping can be applied to reduce communication overhead by mapping communicating processes close to each other.

How the process mapping can be done using a computer is covered in the next section, where we focus on graph theory.

2.3.1 Graph theory

According to [11] mapping can be classified into three broad categories; graph theoretic, mathematical programming and heuristic. The three categories are further described as follows: Mathematical programming approaches the problem by viewing it as an optimization problem and solved using mathematical programming techniques. Heuristic methods provide fast but often sub-optimal solutions within finite time, where an optimal solution cannot

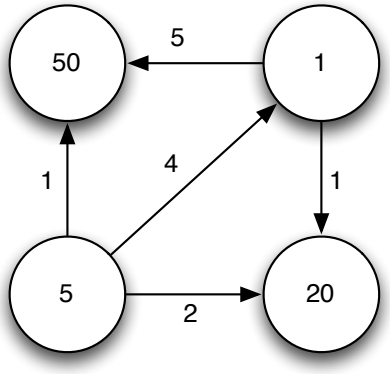


Figure 2.4: Parallel application represented as a graph with weights for communication and processing demands.

be obtained within finite time. We now look at graph theoretic mapping in more detail, as we will use this approach. In graph theoretic mapping, the parallel application can be seen as a graph where each process is a vertex, and each edge is a communication path. The edges and vertices can be weighted. The weight of a vertex refers to the processing demands, while weight on an edge means the communication demands (message count, message size or a combination). The edges can be directed if the direction of the communication is important. This can be the case in for example grids or clusters having full-duplex network interface cards, enabling a certain bandwidth for each direction. Figure 2.4 illustrates a parallel application represented as a directed graph with weights both on the vertices and edges.

In chapter 7 we use Metis [19] as a tool to help us do single node mapping based on intra-node communication, like case c) in figure 2.3. Metis is a powerful library that can be used to partition graphs according weights on vertices, edges or both.

Chapter 3

Related Work

This thesis touches several areas, where three areas stand out. Research on intra-node communication in CMT systems, research done on mapping techniques in different kinds of environments, and queues designed to scale well. In the three following sections we present related work in these areas.

3.1 CMT intra-node communication

Chai et al. [5] have designed and implemented a high performance, scalable MPI intra-node communication scheme for CMP and NUMA systems. By differentiating between small and large messages, transfer overhead is minimized for small messages, and memory usage is minimized for large messages. Intra-node communication latency is improved both for CMP and SMP messages for small and medium sized messages. For large messages, CMP latency goes up. The reason for the increase in latency is not explored, but left to future work.

Chai et al. [6] further present a hybrid approach to intra-node communication, using kernel-assisted memory copy for large messages, and user-space shared memory for small messages. The threshold for small and large messages depends on the case of intra-node communication. Using these *topology aware* thresholds the performance of collective operations is increased by up to 68%. In [4], a technique called data-tiling [15] is used. To fit data into the L2-cache, it is divided in smaller chunks, so that data can fit in the L2-cache and be sent to the neighbor, without going to system memory.

In [8] Fedorova et al. investigate how operating systems should be designed to take advantage of chip multiprocessors for hiding memory latency in workstations. They discover that the shared L2-cache is critical resource in these systems, and propose a cache-conscious scheduling algorithm.

3.2 Mapping

In [18] Leng et al. did a case-study showing that different mapping-strategies have a major impact on performance running the HPL [12] on a cluster of SMP-nodes. Performance is increased by using communication frequency and communication size to find an ideal process to node mapping.

Chai et al. examines the impact of multi-core architectures for cluster computing in [4]. Using benchmarks such as HPL, NAMD [29] and NAS, they find that an average of 50% of communication is intra-node communication in their cluster environment. It is also observed that cache and memory contention can be crucial performance bottle-necks, and propose that middleware and applications should be *multi-core aware*.

Gao et al. [11] describe and implement a graph-matching based task assignment methodology. The method is divided in two parts: First map processes to nodes. When this mapping is done, the second mapping is done on a per-node basis. The entire mapping is done statically, prior to execution of the application. Metis is used to find to find the mapping to use.

3.3 Queues

Darius et al. [3] has presented Nemesis, a scalable, low-latency, message passing subsystem which is integrated with MPICH2 [21]. Scalability and latency are improved by using a single queue for both intra-node and inter-node communication, while intra-node communication is further optimized by using buffers instead of queues.

A *Simple, fast, and Practical Non-blocking* queue based on `compare-and-swap` is presented in [20], and is the inspiration for one of our queue-designs, where only one queue is used (ORQ). `compare-and-swap` was introduced on the IBM System 370, takes as arguments the address of a shared memory location, an expected value, and a new value. If the shared location currently holds the expected value, it is assigned the new value atomically. A boolean return value indicates whether the replacement occured.

Chapter 4

Research platform

This chapter describes the research platform used for the experiments later in the thesis. Two major topics are covered. The first is the hardware which we will run experiments on. The hardware is important both for understanding the results of the experiments, and for understanding the optimizations done. The second major topic is Open MPI, which is the MPI-implementation we will use as a reference, and also improve upon.

The operating system is Linux kernel 2.6.9 running the Rocks Linux distribution [30].

4.1 Hardware

In this section we describe the hardware our experiments run on. Knowing the basic characteristics of the hardware is needed to maximize the performance of our parallel applications, as well as understanding the results of the experiments.

We will run all our experiments on Stallo located in Tromsø. Stallo is currently the second fastest supercomputer in Norway, being rated number

Table 4.1: Stallo hardware specifications

	Aggregated	Per node
Peak performance	60 Teraflop/s	85.12 Gigaflop/s
Nodes	704	1
CPUs	1408	2
CPU cores	5632	8
Memory	12TB	16GB
Network	Gigabit Ethernet + 384 Infiniband	

Table 4.2: Quad-Core Intel Xeon Processor 5300 specifications

Number of Processor Cores	L1-cache (per core)	L2-Cache	Front Side Bus Frequency
4	32KB instruction 32KB data	4MB shared per die 8MB total	1333MHz

61 on the Top500 list June 2008 [34]. Stallo is a cluster of 704 nodes, each having two 2.66GHz Intel Xeon Quad "Clowertown" CPUs for a total of 5632 processing cores. The specifications of the cluster are summed up in table 4.1, and are found at the Stallo documentation homepage [24].

The architecture of Stallo is widely used. While clusters make up for 80% of the Top500 list, the Xeon 51xx (18.2%), 53xx (18.4%), E54xx (23.2%) and X54xx (7.8%) make up for a total of 67.6% of the entire Top500 list.

According to Notur [24], Stallo is intended for:

- Distributed-memory MPI applications with low communication requirements between the processors.
- Shared-memory OpenMP [27] applications using up to eight cores.
- Parallel applications with moderate memory requirements (2-4 GB per core).
- Embarrassingly parallel applications.

Stallo is specifically targeted towards theoretical and computational chemistry to solve problems using Dalton [7] and ADF [31]. Scientific areas include environmental and polar research.

According to [4] 50% of the communication in a parallel application can be intra-node communication, even using only a two-way SMP cluster. The performance of intra-node communication is therefore important. This thesis focuses exclusively on intra-node communication. To eliminate all inter-node communication, we will do all of our experiments using only a single node. We look at the design of such a node in the following section.

4.1.1 Architecture of a Stallo node

Some hardware specifications of the Clowertown are summarized in table 4.2. Each core has its own 32KB private L1-data cache, and L1-instruction cache. The L2-caches that are shared between pairs of cores are 4MB, adding up to a total of 8MB L2-cache per CPU.

As described in chapter 2, the Intel Xeon series of quad-core CPUs are derived from the design of dual-core CPUs. Each quad-core CPU consists of two dual-core CPUs merged together on one chip. As a result each processor core shares cache with one other core, while not sharing any cache with the remaining six cores. Figure 4.1 illustrates two processors residing on a single node in the cluster. Each processor core is labeled by its core ID as reported by the Linux kernel.

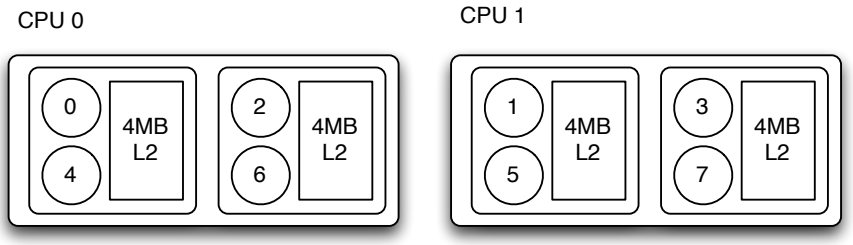


Figure 4.1: Two CPUs mounted on a single node. Each core is labeled as reported by the Linux kernel via `/proc/cpuinfo`.

Performance-wise one can ask the question if such a node should be seen as having two quad core CPUs, or four dual-core CPUs. Since each quad core is designed by merging two dual-cores, one can argue that they perform as four dual-cores.

Another shared resource is the memory bus. The nodes are equipped with a dual independent bus (DIB) which allows each CPU to utilize the full memory bandwidth, regardless of the activity on the other CPU. However, the four cores residing on one CPU are all sharing memory bus. This means each core shares memory bus with three other cores, while not sharing with the remaining four cores.

It should also be noted that the processor cores do not support SMT. In some of the experiments we will run more processes than we have cores (more than 8 processes per node), and it is likely that SMT could have an impact on these experiments. Now that we are fully aware of the design of the quad core processors, we go on to describe the software relevant to our experiments.

4.2 Open MPI

This section gives an overview of the design of Open MPI, and how it handles intra-node, point-to-point communication. In the three following chapters we will propose and evaluate modifications to Open MPI, and it is therefore important to understand the default behavior. The description of Open MPI in this chapter is based on the descriptions in [10, 36], but also reading source code and doing experiments have been needed to get to know the inner details of the modules.

Open MPI is a complete, open-source MPI-1.2 and MPI-2 implementation. It is the successor of LAM/MPI, and is influenced by experience from the LAM/MPI, LA-MPI and FT-MPI projects. It is widely used in high performance computers, and even ships with Mac OS X (since version 10.5).

Open MPI is designed around a component architecture that we will further describe in the next section. Then we will look at how two of the standard components work together to handle intra-node, point-to-point communication.

In the later experiments, we use a default configuration of Open MPI version 1.2.3 compiled with the `--with-platform=optimized` flag, recommended in the installation instructions.

4.2.1 MPI Component Architecture

Open MPI's component-based architecture is designed to be independent of any specific hardware or software environment, and to make it relatively easy to add support for new environments as they become available. To accomplish this, Open MPI defines an MPI Component architecture (MCA) framework that provides services separating critical features into individual components, each with their own functional responsibilities. Individual components can then be implemented as plug-in modules. Each module represents a particular implementation approach (e.g. blocking or non-blocking) or supports a particular environment (e.g. TCP, shared memory, Infiniband etc).

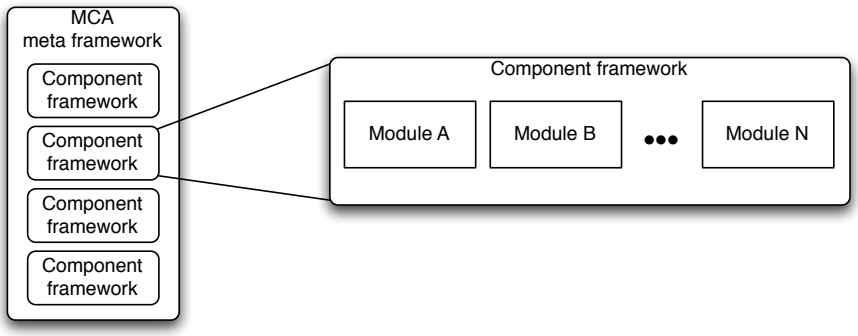


Figure 4.2: The three main functional areas of Open MPI: MCA, component frameworks and the modules for each framework. This figure is based on a figure from [10].

Open MPI has three main functional areas as illustrated in figure 4.2. The three areas are:

- MCA: The backbone component architecture that provides management services for all other layers.
- Component frameworks: Each major functional area in Open MPI has a corresponding back-end component framework that manages modules.
- Modules: Software units that export interfaces that can be deployed and composed with other modules at runtime.

The MCA manages the component frameworks and provides services to them, such as providing the ability to accept runtime parameters from `mpirun` and forward them to the respective framework or module. This allows the user to choose what modules to use at runtime, or specify parameters to tune performance. Another task that the MCA handles, is to find components at build-time and take care of the configuration, building and installation.

A module handles a specific task such as taking care of MPI semantics for collective operations, or moving data between processes at a lower level. The modules are managed by the component frameworks. For example is the Bit Transfer Layer Manager (BML) responsible for choosing what module to use for transferring fragments (messages) between processes. These modules can be based on shared memory or TCP among others.

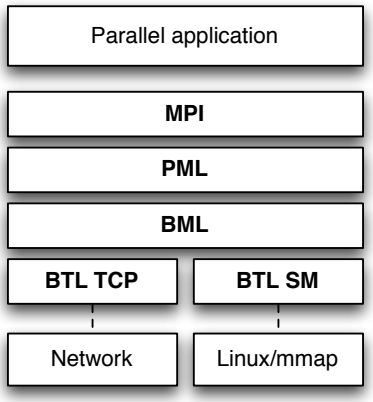


Figure 4.3: The layers involved in point to point communication. In this example we can use the TCP module and the shared memory module (SM) to move the fragments. Either way the semantics of message-passing are enforced by the PML.

Figure 4.3 illustrates the layers involved when doing `MPI_Send` or any other point to point communication. We are now going to give a very short description of the PML and BML before describing how a point to point message pass takes place in the next sections.

- The MPI layer is doing optional parameter checking and down-calls to the PML. MPI-semantic is not handled here.
- Point-to-point management layer (PML) handles all the semantics of the point-to-point calls to MPI. This means fragmentation of messages, ordering and re-assembling the messages. The PML can also distinguish between large or small messages, and handles the semantics of the MPI-calls. For example will the PML make sure that a blocking call such as `MPI_Send` does not return to the application before the message is delivered.
- Bit transfer management layer (BML) is unaware of any MPI-semantics, and only handles transportation of fragments handed to it by the PML module. The BML uses modules such as shared memory (BTL SM) and TCP (BTL TCP). The modules managed by the BML are called BTL-modules which means Bit Transfer Layer.
- The BTL TCP component handles sending of fragments over ethernet networks using the TCP protocol.

- BTL SM handles intra-node communication using shared memory. Only fragments are handled, just as the the other BTL modules. In Linux, `mmap` is used to allocate shared memory regions.

In the following sections we go into the details of an intra-node message pass in Open MPI. We assume the use of the standard modules PML OB1[1] for handling fragmentation of messages (and more), and the BTL SM for transferring fragments between processes. These are the components that provide the main functionality of point-to-point message passing using intra-node communication. To modify this behavior these are the modules that most likely have to change.

4.2.2 Message passing

This section covers how a message is sent from one process to another. We first look at the protocol at a higher level (PML level), and then go down to a fragment level (BTL).

The PML OB1 distinguishes between small and large messages. Small messages are sent eagerly (at once) to reduce latency, while larger messages are synchronized to minimize memory footprint. As the protocol is different between small and large messages, we look at them separately, beginning with eager messages.

Eager messages in Open MPI

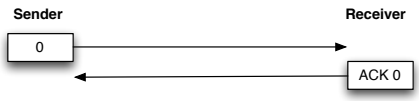


Figure 4.4: Eager message pass between two processes. The sender sends the message in one piece, then gets an acknowledgement in return.

A small message in Open MPI is referred to as an eager message. Using the standard Open MPI configuration, a small message is a message of size 4KB or less. When the sender sends an eager message, the entire message is immediately sent to the receiver. When the receiver receives the message, an

[1]Do not try to make sense of the name (OB1). According to the Open MPI mailing list, the name is inspired from the Star Wars character Obi-Wan Kenobi. There are also other Star Wars inspired names, such as the PML R2 and the BML D2 referring to R2-D2.

ACK is immediately sent back by the BTL component. The message pass is illustrated in figure 4.4. The sender does not wait for the ACK to be received before continuing execution of the program, but gets it the next time it is waiting for a message to be sent or received.

The larger message-passes include synchronization and fragmentation, and are covered in the next section.

Large messages in Open MPI

While the eager messages described in the previous section were sent in one fragment, the larger messages are fragmented as illustrated in figure 4.5. The first fragment is of the same size of the eager fragments (4KB), while the remaining fragments are 32KB. The size of both eager-fragments and the other fragments are configurable by using Open MPI's MCA-parameters.

We will now look at a rendezvous-protocol used for synchronizing two processes. One process is sending while the other is receiving. The rendezvous protocol ensures that the processes communicating are synchronized when the message-pass takes place. Synchronizing the message-pass reduces memory footprint and can prevent problems with resource exhaustion in large cluster environments.

When a large message is to be sent, the sender begins by sending the first small fragment to the receiver. The sender will then wait until the receiver responds to this message with an acknowledgement (ACK). When the sender receives the ACK, the sender will continue sending the rest of the fragments without waiting for ACKs until the entire message has been transferred. The receiver sends an ACK for each fragment received. The protocol is illustrated in figure 4.6.

In chapter 5 we will see how the size of the message and the amount of synchronization influence the performance of communication between cores

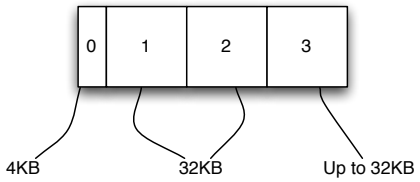


Figure 4.5: A large message divided into four fragments. The first fragment is of the eager message size, while the remaining fragments are of the fragment size.

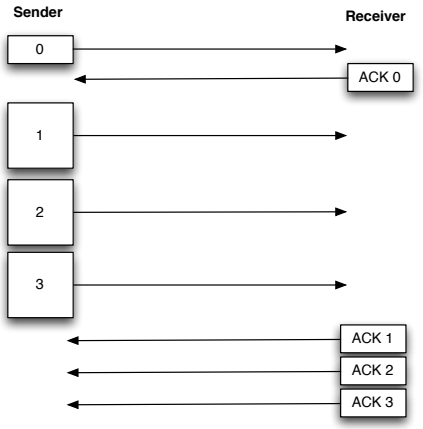


Figure 4.6: Large message-pass between two processes. The sender starts by sending the first fragment of the message as an eager message. When an ACK for that fragment arrives, the rest of the message is sent. The receiver returns an ACK for each fragment.

sharing cache, and cores not sharing cache.

In the next section we move down to the BTL-layer where we look at how fragments are passed between processes in Open MPI. We will in particular look at the shared memory module that handles intra-node communication.

4.2.3 Moving fragments (BTL SM)

The shared memory module SM is a Bit Transfer Layer (BTL) module handled by the Bit Layer Manager (BML). This means that this module will only handle transfer of fragments from one process to another. The fragments may hold a whole message or part of a message. In addition, another module may be responsible for sending other fragments of the same message. The BTL module has two tasks: send the fragments provided by the PML, and reply with an ACK for each received fragment.

The shared memory module transfers fragments between processes using queues stored in user space. In Linux the shared memory is allocated by using the system call `mmap` which maps a file into memory, effectively making a shared memory region for all processes opening this file. The memory mapped area is accessible in the same way as the private address space belonging to the process. It does not require any system calls other than creating and removing the memory region which is done at the initialization and finalization of the application. In the following section we describe the

queues that are used for transferring fragments between processes.

Queues

How the queues are designed, is important for intra-node communication in several ways. The latency of enqueuing and dequeueing data to and from the queue adds to the latency of sending data. There are two important properties of the queues used in BTL SM that we focus on.

The first is that the queues are lock-free, which removes the overhead and complexity of using mutex and locks for protecting the queue when sending and receiving.

The second important property is the number of queues. There are two queues for each pair of processes running on a single node. This makes it possible for several processes to send a message to the same receiver at the same time, as they will write to different queues. In addition, this number of queues allows for the reader and writer of a queue to read and write to it at the same time without locking-mechanisms. Figure 4.7 illustrates the

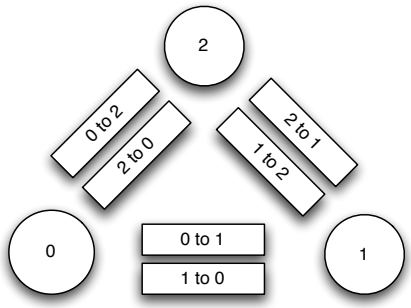


Figure 4.7: Number of queues for three processes.

number of queues used for three processes. For p processes on a single node, there will be $(p-1) \times p$ queues; one receiving queue for each other process, multiplied with the number of processes. This queue-design is similar to the one described in [5].

When the BTL SM is to send a fragment, it will find the right queue using its own and the receiver's MPI-rank, then enqueue the fragment to the queue. When the receiver waits for a fragment, it will spin-wait checking all its $p-1$ receive queues for fragments.

In chapter 6, we will see how changes in queue-design change the scalability of Open MPI.

Chapter 5

Message passing latency

In chapter 4 the architecture of the Stallo-nodes was described. We noticed that the hardware resources shared, varied between pairs of cores. Some cores shared both L2-cache and memory bus. Other cores shared only memory bus, while remaining pairs did not share any resources. In figure 5.1 we have identified three different intra-node communication paths, depending on hardware shared between cores. We have labeled the paths as "intra-die", meaning cores share cache, "inter-CPU" meaning nothing is shared and "inter-die" meaning memory bus is shared, but not cache.

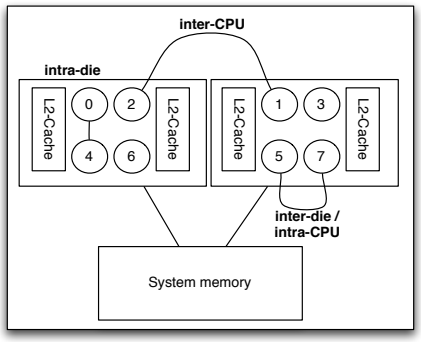


Figure 5.1: Intra-node communication paths, with two quad core CPUs.

In this chapter we measure the performance of these communication-paths. We then demonstrate how performance of "intra-die" communication can be improved when sending large messages, and propose cache-aware message passing to apply the optimal sending strategy for each of the communication paths. In chapter 7 we will also use the results from these experiments, to improve the process to processing core mapping.

Table 5.1: Resources shared between two processes

Case	Shared	Not shared
1	Processing-time, Memory-bus, L1-cache, L2-cache	
2	Memory-bus, L2-cache	Processing-time, L1-cache
3	Memory-bus	Processing-time, L1-cache, L2-cache
4		Processing-time, Memory-bus, L1-cache, L2-cache

5.1 Evaluation

We are now going to evaluate the performance of the intra-node communication paths illustrated in figure 5.1. We expect that the performance will be different for four scenarios:

1. Processes run at the same core, sharing both processor-time, L1-cache and L2-cache.

2. Processes run at two different cores at the same die, sharing 4MB L2-cache.

3. Processes run at cores at same CPU but different dies, each having 4MB L2-cache.

4. Processes run at different CPUs.

The resources shared in the four different cases are summed up in table 5.1. The performance will be evaluated by measuring message passing latency, which we can do by using a ping-pong benchmark. We now describe our ping-pong benchmark, before we evaluate the results of the experiments.

5.1.1 Ping-pong benchmark

To measure the performance of point-to-point message passing we have implemented our own ping-pong micro-benchmark. The benchmark consists of

two processes; one sender and one receiver. The tasks of the sender and the receiver can be described as follows:

Sender	**Receiver**
1. Wait for receiver to be ready to receive a message	1. Wait for and receive message from sender
2. Start timer	
3. Send message	2. Return message
4. Receive message	
5. Stop timer	

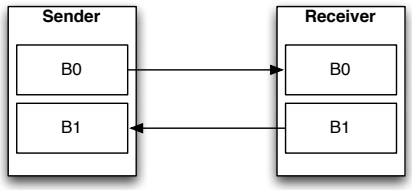
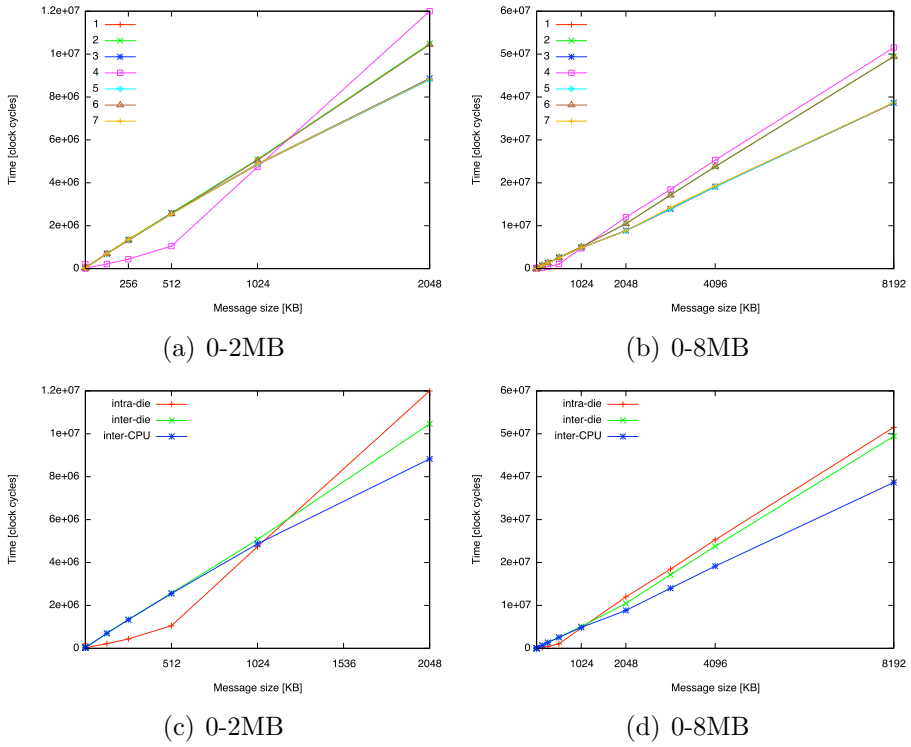


Figure 5.2: Ping-pong message

We have designed the benchmark so that the receiver is always ready to reply to messages from the sender. The sender does both the start timestamp, and stop timestamp, effectively measuring the latency of two message-passes. As illustrated in figure 5.2 the receiver and sender both have two buffers: one for receiving, and one for sending. Both are of the same size. Using two buffers, allows for better benchmark two separate message passes. Using only one buffer, the receiver will already have the reply-message in its cache, before replying. To initialize the underlying Open MPI message passing system, we need to warm up by sending a message before starting the benchmark. This allocates the shared memory queues needed for the message passing.

The sender will always run at core 0, while the receiver will iterate over all the cores from 0 to 7, to measure the latency of all possible communication-paths.



(a) 0-2MB (b) 0-8MB

(c) 0-2MB (d) 0-8MB

Figure 5.3: Time taken for ping-pong message pass. All messages are sent from core 0.

5.1.2 Results

The results for the ping-pong benchmark are presented in figure 5.3. Figure a) and b) contain the results for the message pass from core ID 0 to all other cores, ranging from 1 to 7. In a) we look at the message size from 0 to 2 MB, and b) the message size ranging from 0 to 8MB. Looking at both a) and b), we can easily identify three groups that perform very similar. One group consists of the cores 2 and 6, that can both be categorized as "inter-die/intra-CPU" when sending from core 0. The second group holds 1,3,5 and 7 that all belong to the "inter-CPU" communication path. The last group consists of only core 4, which is the only core sharing cache with 0, and thereby uses the "intra-die" communication path.

The results for the three groups are illustrated in c) and d) for up to 2 and 8 MB respectively. Looking at performance of the three groups up to a message size of 512KB, we see that "intra-die" communication is clearly the fastest, with the others performing at similar levels. From 512KB and up, the "intra-die"-latency increases rapidly relative to the other groups, and also relative to its own latency for smaller message sizes. As the message size

increases above 1MB, the intra-die communication is slower than the other groups, leaving the "inter-CPU" communication as the fastest communication path as the message size increases.

5.1.3 Discussion

When sending large messages, the memory bus clearly becomes a bottleneck for the intra-CPU and intra-die communication paths. For these communication paths the sender and receiver share memory bus. Seeing that the intra-die communication is limited by the memory bus, is surprising as the memory bus is not needed in the communication path; the shared L2-cache should be used for communication in this case. The reason for the L2-cache not being utilized might be explained by the receiver not being able to fetch messages from the receiving-queue fast enough, resulting in the sender overfilling the cache. If the sender overfills the cache, the receiver will have to go to memory to fetch fragments. Remember that the receiver will fetch the fragments in the order they are sent, it has no way of choosing to fetch fragments residing in the cache. We will in section 5.2 present a cache-aware message passing protocol, as a solution to this problem.

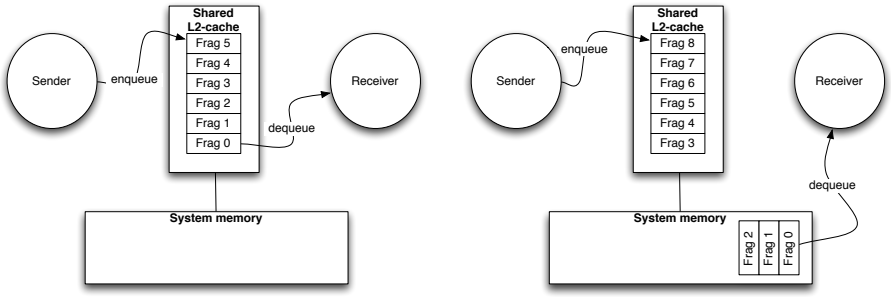
5.1.4 Conclusion

The results of the ping-pong benchmark show that there are two factors important to the latency of message passing. First the location of the cores, basically meaning the hardware shared between sender and receiver, clearly impacts performance. In addition the best communication-path also varied as the message size increased.

For messages less than 1MB, shared cache is the fastest communication-path, while the intra-CPU communication path is fastest for larger messages.

5.2 Cache-aware message-passing

From the previous experiment we learned that the message-passing latency increases faster when the message size reaches a threshold of 512KB for processes sharing L2-cache. We conjecture that the latency increases faster because of the sender overfilling the shared cache, making the receiver go to system memory to fetch the fragments. This is illustrated in figure 5.4 where a) is a message pass where two cores sharing cache are synchronized; the receiver manages to retrieve fragments from cache before the sender overfills



(a) Synchronized (b) Not synchronized

Figure 5.4: Message pass between to processes sharing L2-cache. In a) the processes are synchronized and the system memory is never touched. In b) The sender overfills the cache, forcing the receiver to go to system memory to fetch fragments.

it. In b) the sender overfills the cache, and the receiver will have to go to memory to fetch fragments.

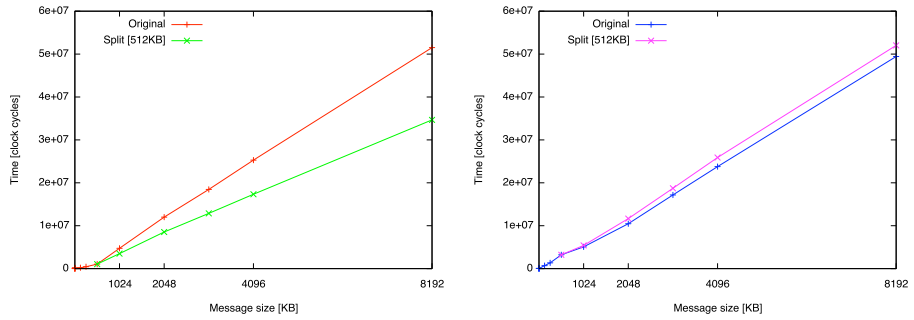
In this experiment we will try to limit the increase in latency by tightening the synchronization used when sending large messages. It is important to notice that this is only true for processes sharing L2-cache. The goal is to have the same latency-growth for messages larger than 512KB as for smaller messages. However, a tighter synchronization requires a certain overhead and we cannot expect to achieve the same performance.

Tightening the synchronization is simply enforced by splitting the message into smaller messages, forcing the rendezvous-protocol described in chapter 4 to synchronize for each 512KB sent. All messages smaller than 512KB are sent as one message. A 1MB message is sent as 2 messages of 512KB, while an 8MB message is sent as 16 smaller 512KB messages. Keep in mind that the messages are still fragmented into 32KB fragments by the PML. We divide the large messages into smaller messages to initiate the synchronization-protocol several times during data transfer.

We evaluate the performance of all intra-node communication paths, and compare them to the results from the last experiment.

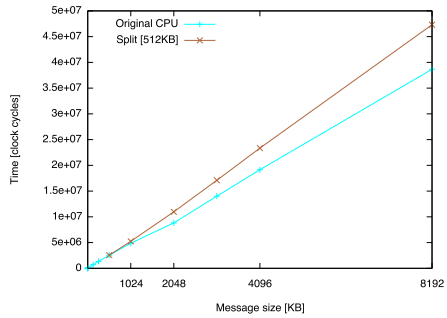
5.2.1 Results and discussion

The results illustrated in figure 5.5 show that tightening synchronization can reduce latency for large messages sent between processes sharing cache. For processes not sharing cache, latency increases. This makes it important that the message passing library can differentiate between processes sharing cache

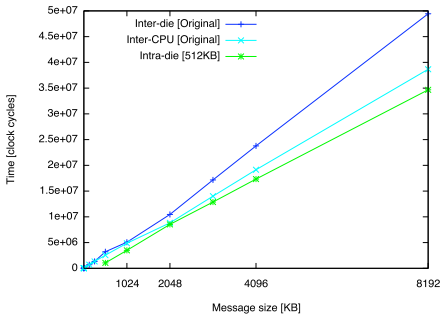


(a) Intra-die

(b) Inter-die



(c) Inter-CPU



(d) Optimal

Figure 5.5: Standard vs stronger synchronization

and processes not sharing cache.

The results illustrated in figure 5.5 compares the results for standard message passing in Open MPI, to our modified approach using stronger synchronization. For shared cache a) we can see that the performance increase dramatically by using the stronger synchronization. For the communication paths not sharing cache, b) and c), the stronger synchronization only adds to the message latency. In d) the best option for all communication paths are illustrated. We observe that the intra-die is now the fastest communication path for large messages as well as small messages, while it was the slowest communication path without the synchronization.

5.2.2 Conclusion

Keeping the sender and receiver synchronized can reduce message passing latency between processes sharing cache. Care should be taken not to add an overhead to messages sent between processes not sharing cache. We have done this in the benchmark application, but it can also be done in the middleware layer, making it transparent to application developers.

Chapter 6

Queue designs

In this chapter we explore the possibilities for improving scalability of the standard Open MPI shared memory implementation. Up until recent years clusters of workstations have mostly been two-way SMP systems, allowing the shared memory queues to focus exclusively on reducing latency for message-passes between two processes. As we see a shift in the industry towards CMT, it is possible that the scalability of intra-node communication system becomes more important than latency between two processes.

In section 4.2 we noticed that the receiver needs to check $p - 1$ queues when waiting for a message, if p is the number of processes. This works well for small scale SMPs where there are typically only two processes, leaving only one queue to be checked. We want to see if it is possible to increase scalability by reducing the number of queues that has to be polled.

We will now present two modifications to the standard Open MPI queues. The first new component uses only one receiving queue per process, and is called "One Receiving Queue" (ORQ). The other is called "Check Only Receiver" (COR) and uses the same number of queues as the standard Open MPI component, but checks only queues where messages are expected to arrive. From now on we will refer to the standard shared memory Open MPI component as SM (the BTL SM component). After the new queue-designs are described, they are experimentally evaluated using the NAS Parallel Benchmarks.

6.1 Design

In this section we describe the design of the COR and ORQ queues. The SM design has already been described in the previous section 4.2. We start with the ORQ, then move on to the COR.

6.1.1 One Receiving Queue

The ORQ component uses only one receiving queue per process as illustrated in figure 6.1. Having only one receiving queue guarantees that the receiver only checks one queue for messages, regardless of the number of processes running at the node. The drawbacks however, are two-fold. When several processes want to send a message to the same receiver (having only one queue), they cannot send the message at the exact same time. The second problem is very much related to the first problem; the queue has to be protected to ensure that the senders do not write to the queue at the same time. The protection adds to the message-passing latency, even when running with a low number of processes. To protect the queue, we have tried two approaches: using a spin-wait mutex, and to use compare-and-swap directly on the queue pointer.

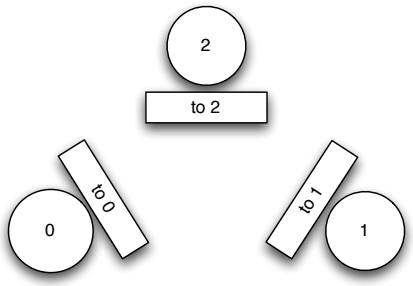


Figure 6.1: Receiving queues for ORQ

6.1.2 Check only receiving queue

The COR component is designed to prevent the senders from competing for receiving-queues, but still keep the number of queues that has to be checked to a minimum.

When doing point-to-point communication using MPI, the sender will specify a receiver rank in the send-call, while the receiver specifies the sender's rank in its receive-call. This means that the receiver always knows from where it expects messages, and does not have to check every queue for incoming fragments. Using Open MPI, we must actually divide the functionality of doing bookkeeping, and the functionality of checking queues according to the MCA described in chapter 4.2. We divide them into following components:

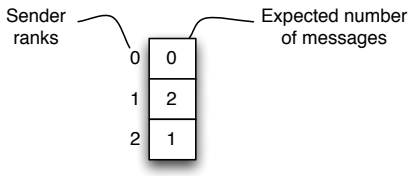


Figure 6.2: Structure keeping count of number of expected messages from each other process.

1. PML COR - This is where we keep track of expected messages. This adds to latency.

2. BTL COR - This is where we can use information provided by the PML COR to only poll queues where we expect fragments.

In the PML the receiver can do this bookkeeping by keeping a one-dimensional array of size $p - 1$. Each element in the array is initialized to 0 when the application starts. Whenever a variation of a receive-call is made, a counter in the array is increased. The sender is the index to the array, where the array holds the number of messages expected. When a message is received, the counter can be decreased. Figure 6.2 illustrates a structure holding the message-counters.

In the BTL we can reduce the number of queues we have to pull, by only polling the queues where the count is larger than 0.

There is however one MPI-feature that complicates the matter somewhat. The MPI-standard allows the receiver (not the sender) to specify a wild card (`MPI_ANY_SOURCE`) instead of a specific sender rank. This wild card means that a message will be accepted from anyone. To deal with this, an extra counter is made. If a message is expected from any source, this counter is increased. Whenever a message is received, the any source counter is decreased only if we did not expect a message from the sender.

The sender does no bookkeeping in any case, and behaves as in the standard SM implementation. In the next section we evaluate the three components, SM, ORQ and COR.

6.2 Evaluation

We are now going to evaluate how the different queue-designs described in the last section affect the performance of some of the NAS Parallel Benchmarks.

The queues we suggest are designed to improve Open MPI's intra-node scalability, and we will therefore do the experiment with a large number of processes, up to 128. Using a single node, this means that each of the eight cores will be running up to 16 processes each. The details of the nodes can be found in chapter 4.

6.2.1 Results and discussion

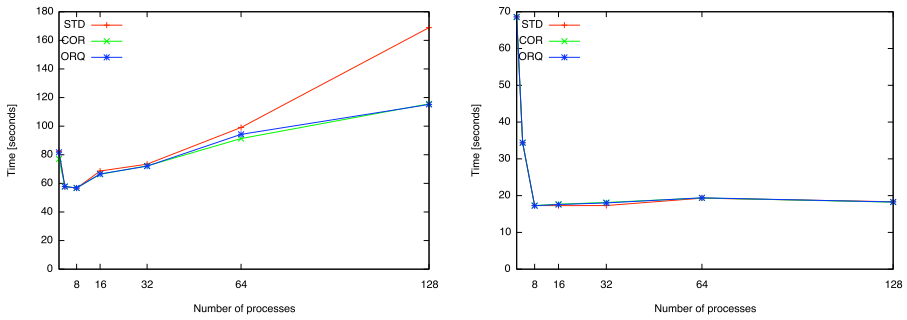
Figure 6.3 illustrates the performance of the three different components, for six of NAS Parallel Benchmarks. The first thing to notice is how the execution-time decreases from 2-8 processes. This is because we have eight available processor-cores that are utilized as we increase the number of processes up to 8. When we further increase the number of processes (but not the number of processing cores), the execution time increases.

Looking at the execution times for 128 processes, the ORQ and COR components had the lowest execution-time in three benchmarks each. The SM was never fastest. However, the ORQ component was slower than both COR and SM in the IS-benchmark. Table 6.2 summarizes the count for fastest and slowest for each of the components.

Table 6.1: Benchmark results for 128 processes

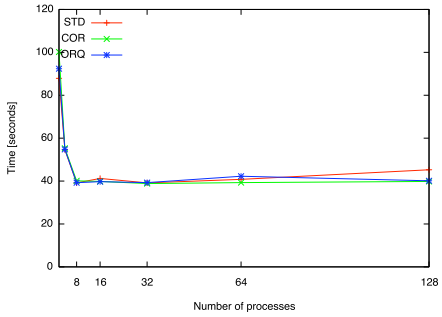
Bench-mark	Execution time			Performance increase from slowest to fastest
	STD	COR	ORQ	
CG	168.9s	115.62s	115.18s	31.81%
EP	18.33s	18.23s	18.27s	0.55%
FT	45.26s	39.82s	40.13s	12.02%
IS	3.89s	3.58s	5.16s	30.62%
MG	18.11s	13.32s	13.09s	27.72%
LU	179.6s	159.8s	156.6s	12.8%

Looking at table 6.1 we can see that the performance difference between the fastest and slowest component was ranging from no improvement to 32%. The biggest improvement was the CG benchmark that utilizes a high number of point-to-point communication. The smallest improvement was the EP (embarrassingly parallel problem) that hardly communicates during the runtime of the application.

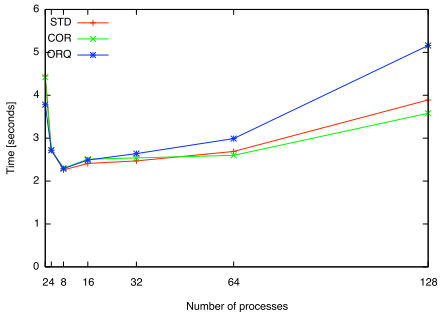


(a) CG

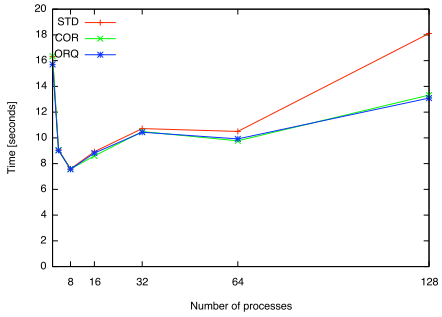
(b) EP



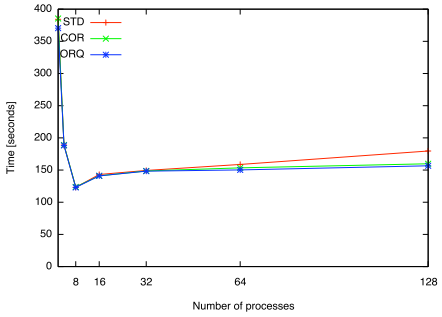
(c) FT



(d) IS



(e) MG



(f) LU

Figure 6.3: Execution time for some of the NAS Parallel benchmarks using different queue-designs.

Table 6.2: Fastest and slowest component for 128 processes.

Component	Fastest	Slowest
SM	0	5
COR	3	0
ORQ	3	1

6.2.2 Conclusion

We have demonstrated that reducing the number of queues that need to be polled, can greatly improve scalability of Open MPI, reducing execution-time by certain applications by up to 30%. Observing that senders in ORQ must compete for receiving queues, we suggest COR that reduces number of queues to be polled, while avoiding conflict between senders.

By using knowledge of where messages are expected to arrive, we can reduce the number of queues that need to be polled, without having senders competing for receiver-queues.

Chapter 7

Single node process-to-core mapping

We give an overview over mapping in various systems in chapter 2. In this chapter we focus exclusively on mapping on a single node. The goal is to reduce execution time of parallel applications, by utilizing the shared hardware resources. The architecture of the node used is described in chapter 4.

We will do three experiments in this chapter. The first is an experiment where we map parallel applications that only utilize four of the node's eight cores. From chapter 5 we learned that communication is faster between cores sharing L2-cache. In this experiment we want to see what gives the greater performance benefits; sharing L2-cache for potentially faster communication, or having more private cache.

The second experiment is a case study of the CG NAS Parallel benchmark. The goal is to find a mapping where the communication overhead of the benchmark is minimized. This means moving communicating processes to cores sharing L2-cache. The mapping is static, and decided prior to running the benchmark.

In the third experiment we try to achieve the performance benefits gained with static mapping, without analyzing the applications communication patterns prior to running the application. We design a dynamic mapper that analyzes the communication pattern during runtime, and uses Metis to find a good mapping for the application.

7.1 Mapping to a subset of available cores

In this experiment we explore how we should map a parallel application that runs only on a single node, and uses only a subset of the available cores. In

chapter 5 we learned that message passing is faster between two processes sharing L2-cache. We also know that processes sharing cache can pollute each others cache and have to share a single memory bus. Mapping the application to cores not sharing cache will give each process a private cache, and can utilize both memory busses.

The main goal of this experiment is to examine if it is better to share cache to reduce communication-time, or if it is better to have separate caches. In addition to share cache or not, we see how the benchmarks perform when sharing memory bus. We will use four of the eight available cores in this experiment. As with the message passing latency, we have three choices when mapping processes to cores:

1. Map all four processes to one CPU. Each pair of processes can communicate faster, but pollutes each others cache and utilize only one memory bus.

2. Map each pair of processes to one die. Each pair of processes share L2-cache, but have their own independent memory bus.

3. Use one core at each die. Each core gets a private L2-cache. We also utilize both memory buses.

The grey cores in figure 7.1 illustrate the cores used in each of the three cases.

We will use the NAS Parallel Benchmarks as benchmarks in this experiment, and in the next section we look at how they perform as we use different mappings.

7.1.1 Results and discussion

Table 7.1 contains the execution times for the various NAS Parallel Benchmarks run in the three different cases described above. The cases are identified by what cores they run at. 0,2,4,6 means that processes run at the same CPU. 0,1,4,5 means that we run two pairs sharing cache, but at different CPUs, and 0,1,2,3 means one process at each die.

For every benchmark run, we see that the lowest execution time is with the case 3 mapping. The case 3 mapping means that each process has its own L2-cache, and both memory buses are utilized. Only for the EP (embarrassingly parallel problem) is the mapping not significant.

While case 1, where L2-cache and memory bus are shared, is the slowest case, the performance jump between case 1 and case 2 is bigger than the jump between case 2 and case 3. The only difference between these two cases, is

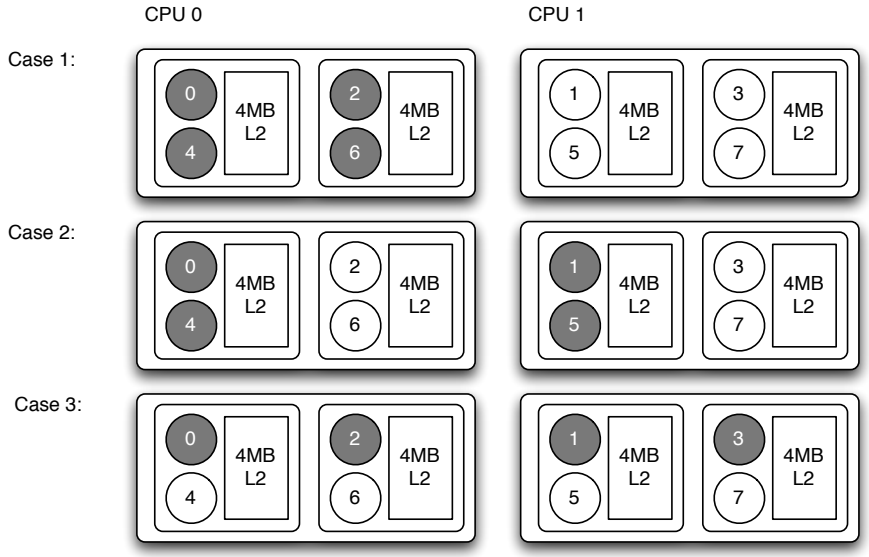


Figure 7.1: The three cases for mapping four processes to cores.

the memory bus; indicating that the memory bus is limiting performance more than the shared L2-cache.

Table 7.1: Execution time of the NAS Parallel Benchmarks utilizing 4 cores.

Benchmark	Case 1 (0,4,2,6)	Case 2 (0,4,1,5)	Case 3 (0,1,2,3)
BT	345.11s	310.25s	307.46s
SP	443.44s	313.80s	272.75s
CG	83.60s	63.35s	57.77s
EP	34.42s	34.47s	34.45s
FT	67.51s	55.34s	55.01s
MG	15.80s	10.31s	9.15s

7.1.2 Conclusion

The way a parallel application is mapped to a subset of the eight cores on a single node, does influence the performance of applications such as the NAS Parallel Benchmarks. All benchmarks have the shortest execution time when mapped in such a way that each process has its own private L2-cache, and

both memory busses are used. Also the memory bus appears to be limiting performance more than the L2-cache.

The Linux kernel identifies the cores in such a way that a round-robin mapping should be the optimal for most applications. The exception is applications that can utilize the shared cache like we did in with the message passing latency experiment in chapter 5.

7.2 Static mapping

In this experiment we explore the potential for reducing execution time for a real life application, using what we have learned about the application, communication and the cache:

- In chapter 5 we learned that communication can be faster between cores sharing L2-cache.
- From section 7.1 we learned that real life applications utilizing four cores, are better off not sharing cache and rather use the full memory-bandwidth and twice the amount of cache.

Based on this we know that it might be possible to decrease communication overhead by letting communicating processes run on cores sharing L2-cache. This experiment is a case study of the CG NAS Parallel Benchmark. The CG benchmark does a high amount of point-to-point communication [1], and has a communication pattern that is easy to illustrate and reason about. Point-to-point communication makes it easier to study the communication pattern, and the lack of global communication also makes it more likely to find a good mapping.

7.2.1 Communication pattern

The communication pattern of the CG NAS Parallel Benchmark is illustrated in figure 7.2. Each process is represented by a circle and identified by its MPI-rank. The communicating processes are connected by dotted or solid lines. The solid lines represent a higher number of messages sent than the dotted line. Focusing only on the solid lines, we can see that it is easy to partition this application into four partitions.

As we know, each Stallo node has 8 cores. However, in this experiment it is the number of shared L2-caches that is interesting. We want to partition the communication graph in such a way that communicating processes are mapped to cores sharing cache. We have four shared L2-caches which means

that we want each of the four groups mapped to cores sharing L2-cache. Table 7.2 shows a possible mapping for process to cores. It is important that we let each process use any of the two cores sharing the same L2-cache. If we restrict each process to one core, we waste potential for increasing the thread level parallelism (TLP).

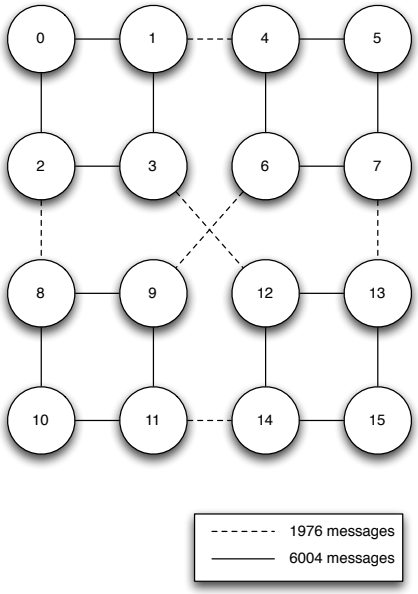


Figure 7.2: Communication pattern of the CG benchmark using 16 processes and size class B.

Table 7.2: Possible mapping for the CG benchmark.

Process groups (MPI-rank)	Pairs of cores sharing L2-cache (Core ID)
0,1,2,3	0,4
4,5,6,7	1,5
8,9,10,11	2,6
12,13,14,15	3,7

7.2.2 Results and discussion

The results in figure 7.3 and table 7.3 show that we are not able to reduce the execution time of the application, by mapping communicating processes closer to the same L2-cache when running one process for each core (8 processes). When increasing number of processes per core to 2 and 4, we see that the static mapping increases performance by up to 11%. When increasing number of processes to 8 per core (64 per node), the performance improvement is less than 5%. The reason that the improvement for 64 processes is smaller than for 32 processes, is probably that the communication pattern changes so that it does not partition as nicely to the four caches.

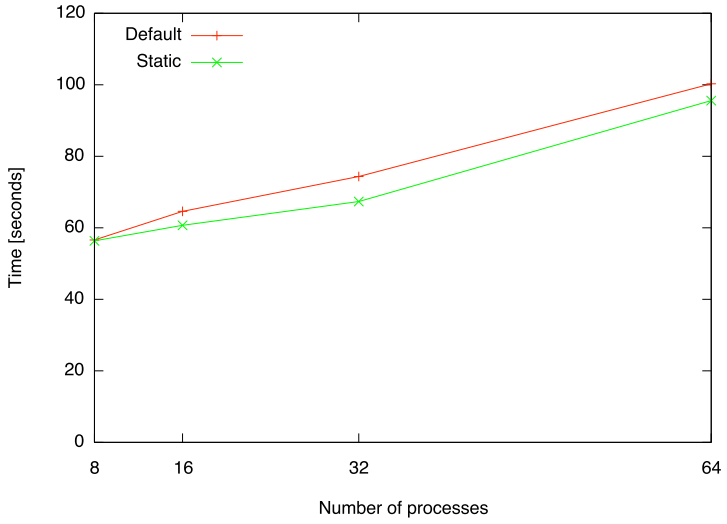


Figure 7.3: Performance of the CG NAS Parallel Benchmark using static mapping versus default mapping

Table 7.3: Static versus default mapping

Processes	Default	Static	Improvement
8	56.7s	56.4s	0.6%
16	68.6s	60.7s	11.6%
32	73.4s	67.4s	8.3%
64	100.1s	95.6s	4.5%

7.2.3 Conclusion

By analyzing the CG benchmark and finding a better mapping for it, we managed to reduce execution time of the application when running more than one process for each core, but not when running one process per core. Since decomposing applications with one process per core is the standard approach, and in this case also the fastest, doing mapping of applications is not beneficial. When overdecomposing [1] the application, we are able to increase performance by up to 11% for two processes per core, compared to the default mapping.

7.3 Dynamic mapper

The goal of the dynamic mapper is to provide the performance-benefits of the static mapper, without the disadvantage of static mapping. The disadvantage of static mapping is that the mapping has to be done manually by someone who knows both the communication pattern of the application and the underlying environment the application will run in.

The dynamic mapper should find a good mapping for any application using knowledge of the hardware (configured by the system administrator) and analyzing the critical traffic of the application during runtime. Even though we do not need any prior knowledge of the application, we do need some information about the system architecture. We need to know what cores that can communicate faster between them. In our experiment running on the Stallo-node, core-pairs `[0,4]`, `[1,5]`, `[2,6]` and `[3,7]` can communicate faster between themselves as we discovered in chapter 5.

We also know that if the application does not utilize all cores, it should be spread to utilize the memory bandwidth and cache.

In the next section we will look at the design of a dynamic mapper.

7.3.1 Design

When designing dynamic mapping for Open MPI, we need to answer the following questions:

1. What data to log and base our mapping on?
2. Where and when to evaluate the mapping during runtime?
3. How to find a good mapping?

How we deal with these questions is described in the following three sections.

What data to log

All MPI-based communication is available for us to log in the MPI-layer. This includes point-to-point messages as well as collective operations. In addition we know the size of the messages being sent.

We use the message-passing count to decide which processes that communicate most, and ignore message size. In addition we have limited logging to point-to-point communication.

Where and when to evaluate the mapping

There are three places this remap can take place:

1. In Open MPI's progress function for one process at each node.

2. Spawning a mapper-process at each node, taking care of the mapping at given time intervals

3. Make a new MPI-function (e.g. `MPI_Remap()`), letting the application developer decide when to remap the processes.

The two first alternatives are hidden from the application programmer, and can actually be combined with the third option, the MPI-function.

The advantage of the first alternative, is that we do not need to spawn another thread per node. The disadvantages are that it can be difficult to predict when the progress function will invoke the evaluation of the mapping, because the process might be doing other work. In addition the mapper would be restricted to the core(s) that the process is running on.

The advantages of spawning a new process taking care of the mapping, are that it is quite easy to predict when it will run, and it can run on any available core. The disadvantage is oversubscribing the cores. However, the mapping-process should be sleeping most of the time.

Making a new MPI-function puts the application developer in control of when the mapping should take place. This way it is possible to combine the application developer's knowledge of the application, while using Open MPI's logged data.

Our dynamic mapper is run by spawning a mapper-process at each node, taking care of the mapping. This way the mapping is transparent to the application developer, and we can easily control how often the mapper should run to re-evaluate the mapping.

How often to evaluate the mapping

By using an independent mapper process, we can use the sleep-primitive to make the mapper run at certain time intervals. When the mapper wakes up, we can analyze the data logged since last time.

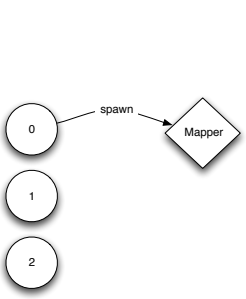
How to find a good mapping

The theory of mapping is discussed in chapter 2, and a quick introduction to graph-theory in mapping is given. In graph-terminology, what we want to do is to partition the graph into partitions. Each partition should consist of the processes communicating, and should run on the cores sharing cache.

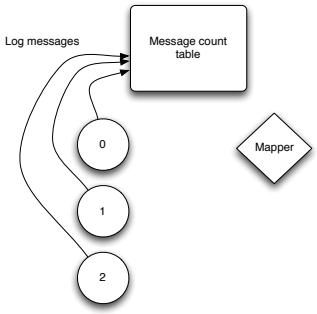
There are several tools for doing this partitioning such as Metis and Scotch [32]. We have decided to use Metis in this project as it provides the functionality we need, and has been used earlier for doing static partitioning. Metis provides the functionality to partition weighted graphs, taking the graph as an argument along with the number of wanted partitions. The return value is an array holding the optimal partitioning for the given graph.

Summary

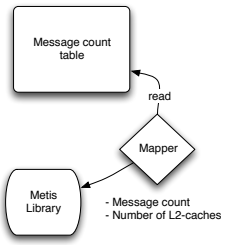
Figure 7.4 describes the workings of the dynamic mapper process. At each node (we only use one), a mapper-process is spawned by the process with the lowest mpi-rank. This happens during `MPI_Init`. All processes log all sent point-to-point message passes (`MPI_Send` and its variations) in a table stored in shared memory. The mapper reads the table, and creates a graph that is passed to Metis for partitioning. When Metis returns the partitioning, the mapper enforces it with the `sched_setaffinity` function.



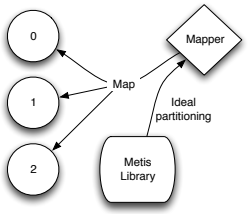
(a) One process at each node spawns a mapper-process.



(b) The processes logs each message-pass in a table shared in memory.



(c) Mapper feeds Metis with data read from message-count table.



(d) The mapper maps processes according to Metis' partitioning.

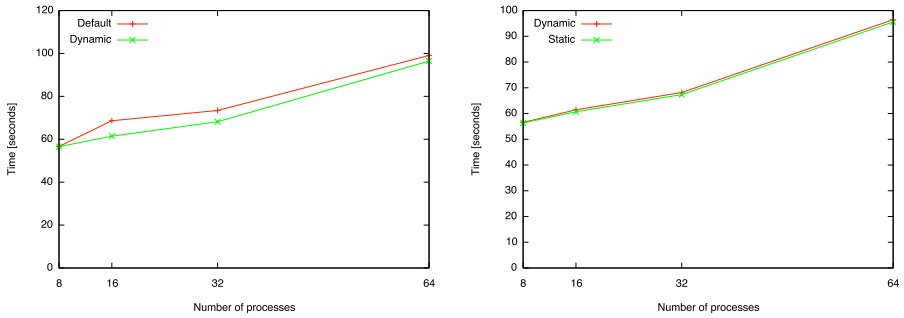
Figure 7.4: Dynamic mapper.

7.3.2 Evaluation of dynamic mapper

In this section we evaluate the performance of the dynamic mapper and compare the results to the static mapping. We evaluate both the CG benchmark that was evaluated for static mapping, and in addition evaluate the performance of the dynamic mapper used on the SP and BT benchmarks.

Table 7.4: Dynamic versus default mapping

Processes	Default	Dynamic	Improvement
8	56.7s	56.4s	0.2%
16	68.6s	60.7s	10.5%
32	73.4s	67.4s	7.2%
64	100.1s	95.6s	3.6%

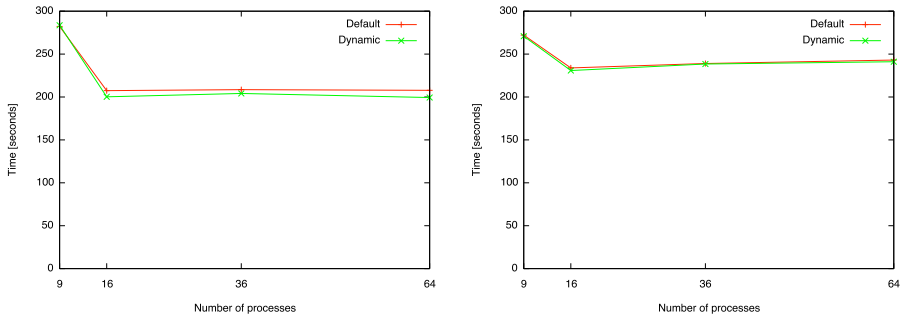


(a) CG (b) Static vs dynamic

Figure 7.5: Execution time for CG using the dynamic mapper.

The results in table 7.4 illustrated in figure 7.6 a) show that we can achieve much of the benefits from static mapping shown in section 7.2 without prior knowledge of the application's communication patterns. In b) the performance of the static and dynamic mapper are compared.

The results from other benchmarks shown in figure 7.6 illustrate that a reduction in execution-time is very dependent on the application pattern, and how well it maps to the preferred number of partitions. However, the mapper does not seem to add a high overhead to the system, making it possible to leave it running even when the mapping does not increase performance.



(a) BT (b) SP

Figure 7.6: Execution time for two other NAS Parallel Benchmarks using point-to-point communication

7.3.3 Conclusion

Using dynamic mapping we can achieve most of the performance gain from static mapping. This is done by configuring the dynamic mapper (in middleware) to the environment it runs in, regarding cores sharing cache. The dynamic mapper is transparent to the user, and has a low overhead, making it a good alternative to static mapping.

From the experiment using only four of eight cores in section 7.1 we learned that parallel applications should be mapped to utilize all hardware resources (all L2-caches and both memory busses). This should also be taken care of in the dynamic mapper, so that it does not map processes close to each other when the application should be spread to utilize all L2-caches and both memory busses.

Chapter 8

Conclusion and future work

In this thesis we have examined performance-characteristics of the chip-multiprocessors widely used in today's supercomputers. We have identified problems with the Open MPI implementation of MPI regarding scalability and lack of optimization to the underlying hardware-architecture. We have proposed improvements to increase scalability, introduced mapping of processes to processor cores, and optimized message passes for messages sent between cores sharing cache.

Intra-node communication depends heavily on the communication-path (i.e. the amount of shared hardware). For small messages, communication between processes sharing cache is fastest. For large messages, processes using both memory busses are fastest. We increase the performance of large messages sent between cores sharing cache by keeping the sender and the receiver synchronized during the entire transfer. This enables the receiver to access the receiving queue in cache, rather than fetch it from memory. Using the improved protocol for shared cache, message transfer between cores sharing cache is the fastest for both small and large messages. For processes not sharing cache, the increased synchronization is an added overhead, so care should be taken to only apply increased synchronization on messages sent between processes sharing cache.

Scalability is improved by using two new queue-designs, reducing the number of queues that needs to be polled when a high number of processes run on a single node. For most applications, using only one queue (ORQ) is the fastest, but we notice that using only one queue can reduce scalability when senders have to compete for the queue. By doing bookkeeping (COR) at the receiver side of message passes, we are able to reduce the number of queues that needs to be polled, without having the senders compete for queues. It should be noted that while the COR queue-design is good for all NAS Parallel benchmarks, it will be no better than the standard queue-

design, if the MPI_ANY_SOURCE wildcard is used to receive messages from any process. The receiver then needs to check all queues. Based on the NAS Parallel benchmarks, the COR queue-design is the best trade-off between checking fewer queues, and prevent competition when sending.

We were not able to improve performance of the CG benchmark when doing manual mapping using one process for each core on a single node. When increasing the number of processes per core to 2 and 4 (16 and 32 on a node), our mapping performed up to 10% better than the default mapping with the same number of processes. The best performance was still measured using one process per core, and any mapping. The dynamic mapper performed comparable to the static mapper, increasing performance relative to the default mapping when running 16 or 32 processes on a single node. The dynamic approach to mapping is more attractive than the static, as it performs comparable without needing to analyze the applications communication pattern before running the application.

The presented techniques have given performance increase in different scenarios; the dynamic mapper for one application when oversubscribing, the queues when running a high number of processes, and the cache-aware message passing protocol for large messages. We can learn from this, that minimizing intra-node communication overhead, as many other problems, has to be approached from several angles. Each optimization must be carefully constructed not to increase significant overhead in the common cases. Combining techniques similar to those presented in this thesis, with other techniques, is necessary to benefit from the parallel hardware we have today and in the future. Further more, component based communication middleware can be a powerful tool for optimizing performance to the available hardware, as it can behave dynamically at execution or even during runtime.

Future work

All queue designs explored have a weakness; either by requiring the receiver to poll many receiving queues, or to have the senders compete for queues. The goal of a good queue-design is to find a balance between few queues for the receiver to poll, and enough queues that the senders do not have to compete to send messages. A variation of the ORQ component using more than one queue, maybe combined with bookkeeping as in COR, can be a good approach.

For communication between processes sharing cache, we managed to increase performance by increasing synchronization. An approach where the sender and receiver are synchronized with a smaller overhead would be desirable. Having the sender keep track of number of fragments in the queue

might be another way of keeping synchronization with less overhead.

The dynamic mapper should be tested with more applications to see whether it is possible to increase performance of applications when running one process per core.

Bibliography

[1] L. A. Bongo, B. Vinter, O. J. Anshus, T. Larsen, and J. M. Bjørndalen. Using overdecomposition to overlap communication latencies with computation and take advantage of SMT processors. In *proceedings of ICPPW '06: The 2006 International Conference Workshops on Parallel Processing*, pages 239–247, Columbus, OH, USA, August 2006. IEEE Computer Society.

[2] D. Buntinas, G. Mercier, and W. Gropp. Data transfers between processes in an SMP system: Performance study and application to MPI. In *proceedings of ICPP '06: The 35th International Conference on Parallel Processing*, pages 487–496, Columbus, OH, USA, August 2006. IEEE Computer Society.

[3] D. Buntinas, G. Mercier, and W. Gropp. Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem. In *proceedings of CCGRID '06: the Sixth IEEE International Symposium on Cluster Computing and the Grid*, pages 521–530, Singapore, May 2006. IEEE Computer Society.

[4] L. Chai, Q. Gao, and D. K. Panda. Understanding the impact of multi-core architecture in cluster computing: A case study with Intel dual-core system. In *proceedings of CCGRID '07: The Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 471–478, Rio de Janeiro, Brazil, May 2007. IEEE Computer Society.

[5] L. Chai, A. Hartono, and D. K. Panda. Designing high performance and scalable MPI intra-node communication support for clusters. In *CLUSTER '06: Proceedings of the 2006 IEEE International Conference on Cluster Computing*, Barcelona, Spain, September 2006. IEEE Computer Society.

[6] L. Chai, P. Lai, H.-W. Jin, and D. K. Panda. Designing an efficient kernel-level and user-level hybrid approach for MPI intra-node commu-

nication on multi-core systems. In *In proceedings of ICCP' 08: the 37th International Conference on parallel processing*, pages 222–229, Portland, OR, USA, September 2008. IEEE Computer Society.

[7] Dalton. `http://www.kjemi.uio.no/software/dalton/dalton.html`.

[8] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *proceedings of ATEC '05: the annual conference on USENIX Annual Technical Conference*, pages 395–398, Anaheim, CA, USA, April 2005. USENIX Association.

[9] Folding@home homepage. `http://folding.stanford.edu/`.

[10] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *proceedings of the 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[11] H. Gao, A. Schmidt, A. Gupta, and P. Luksch. Load balancing for spatial-grid-based parallel numeric simulations on clusters of SMPs. In *proceedings of EuroPDP '03: the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 75–82, Genova, Italy, February 2003. IEEE Computer Society.

[12] High Performance Linpack. `http://www.netlib.org/benchmark/hpl`.

[13] Intel. `http://www.intel.com`.

[14] Q. Jacobson. UltraSPARC IV Processors. In *proceedings of the Microprocessor Forum*, San Jose, CA, October 2003.

[15] I. Kadayif and M. Kandemir. Data space-oriented tiling for enhancing locality. *TECS: ACM Transactions on Embedded Computing Systems*, 4(2):388–414, May 2005.

[16] S. Kapil. UltraSPARC Gemini: Dual CPU processor. In *proceedings of the 15th annual Hot Chips Symposium*, Stanford University, Palo Alto, CA, August 2003.

[17] P. Kongetira. A 32-way multithreaded sparc procesor. In *proceedings of the 16th annual Hot Chips Symposium*, Stanford University, Palo Alto, CA, August 2004.

[18] T. Leng, R. Ali, J. Hsieh, V. Mashayekhi, and R. Rooholamini. Performance impact of process mapping on small-scale SMP clusters - a case study using high performance Linpack. In *proceedings of IPDPS '02: the 16th International Parallel and Distributed Processing Symposium*, page 263, Fort Lauderdale, FL, USA, April 2002. IEEE Computer Society.

[19] Metis: Serial graph partitioning. `http://glaros.dtc.umn.edu/gkhome/views/metis/`.

[20] M. M. Michael and M. L. Scott. Fast and practical non-blocking and blocking concurrent queue algorithms. In *proceedings of PODC '96: The 15th ACM Symposium on Principles of Distributed Computing*, pages 267–275, Philadelphia, PA, USA, May 1996. ACM.

[21] MPICH2. `www.mcs.anl.gov/mpi/mpich/`.

[22] NASA. NAS parallel benchmark suite homepage. `http://www.nas.nasa.gov/Resources/Software/npb.html`.

[23] Niagra. `http://www.sun.com`.

[24] Notur. Stallo documentation homepage. `http://www.notur.no/hardware/stallo/`.

[25] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *proceedings of ASPLOS-VII: The seventh international conference on Architectural support for programming languages and operating systems*, pages 2–11, New York, NY, USA, October 1996. ACM.

[26] Open MPI homepage. `http://www.open-mpi.org/`.

[27] OpenMP group. `http://www.openmp.org/`.

[28] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *proceedings of SC '03: The 2003 ACM/IEEE conference on Supercomputing*, Phoenix, AZ, USA, November 2003. IEEE Computer Society.

[29] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten. Scalable molecular dynamics with NAMD. In *Journal of Computational Chemistry*, pages 26:1781–1802, 2005.

[30] Rocks cluster. `http://www.rocksclusters.org/`.

[31] Scientific computing and modeling. Amsterdam density functional package. `http://www.scm.com/`.

[32] Scotch. `http://www.labri.fr/perso/pelegrin/scotch/`.

[33] L. Spracklen and S. G. Abraham. Chip multithreading: Opportunities and challenges. In *proceedings of HPCA '05: The International Symposium on High Performance Computer Architecture*, pages 248–252, San Francisco, CA, USA, February 2005. IEEE Computer Society.

[34] Top500 supercomputer sites. `http://www.top500.org/`.

[35] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *proceedings of ISCA '95: The 22nd Annual International Symposium on Computer Architecture*, pages 392–403, Santa Margherita Ligure, Italy, June 1995.

[36] T. S. Woodall, R. L. Graham, R. H. Castain, D. J. Daniel, M. W. Sukalski, E. Gabriel, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, P. Kambadur, B. Barrett, and A. Lumsdaine. Teg: A high-performance, scalable, multi-network point-to-point communications methodology. In *proceedings of the 11th European PVM/MPI Users Group Meeting*, pages 303–310, Budapest, Hungary, September 2004.