Großer Beleg

MPICH on Fiasco.OC/L4Re

Michael Jahn

16. April 2013

Technische Universität Dresden
Faculty of Computer Science
Institute of Systems Architecture
Chair Of Operating Systems

Supervising Professor: Prof. Dr. rer. nat. Hermann Härtig
Supervisor: Dipl.-Inf. Carsten Weinhold

Selbstständigkeitserklärung (Statutory Declaration)

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 16. April 2013, Michael Jahn

Acknowledgements

First I want to thank my supervisor Carsten for guiding me through the process of finishing this work. Further thanks are due to everyone at the OS chair and in the OS lab who kindly helped me with advise, insight and hardware. I want to thank my friend Dirk for lending me an ear when I encountered problems in my work. Last but not least... thanks go out to my dearly beloved Suse and our son Gustav for indulging my coming home late and bringing happiness into my life.

Contents

List of Figures

1 Introduction

MPICH is a widely used implementation of the Message Passing Interface (MPI) specification. Fiasco.OC is a microkernel of the L4 family developed at the OS Chair of TU Dresden. The L4 Runtime Environment (L4Re) constitutes the most basic part of its user-land. Together Fiasco.OC and L4Re form an object-capability system that offers basic primitives and services to applications.

In my work I will show that MPI applications can be compiled for and run on top of Fiasco.OC/L4Re and port a simple process manager to launch MPI processes utilizing shared-memory for message passing.

Chapter 2 introduces into the terminology required to understand this work. I will explain basic aspects of the message-passing parallel programming paradigm. Further I explicate the MPI message-passing specification and the MPICH implementation of MPI. I will give a short introduction how MPI applications are set up into processes by process managers. In Chapter 3 design and implementation aspects of libraries that were required to create a working port are discussed. I will benchmark my port with regard to one- and two-sided operations in Chapter 4 and compare its performance with MPICH on Linux. I will also discuss differences of both systems and their effect on performance. My work finishes with concluding remarks and hints to future work.

2 Background

2.1 Parallel Computational Models

Parallel Computational Models (PCM) differ in their use of memory, e.g if it is physically shared or distributed, if memory access triggers hardware or software communication and what is the unit of execution.

In this section I will first give a short overview of different PCMs and then detail relevant models for the Message Passing Interface. This section is based on *Using MPI, 2nd Edition: Portable Parallel Programming with the Message Passing Interface* [GLS99, p. 2-11].

2.1.1 Data Parallelism

The term data parallelism already hints that the parallelism of this PCM comes entirely from the data. On the hardware side vector computers are typical exponents of this. Vector computers execute an operation with one or more vector-operands in a single instruction, that means they may add two vector-registers into a result vector-register using only one instruction. Data parallelism is equivalent to Single Instruction Multiple Data (SIMD) processing in Flynn's taxonomy [Fly72]. SIMD instructions can be found in todays off-the-shelf CPUs. The Streaming SIMD Instructions (SSE) and Advanced Vector Extensions (AVX) used on Intel and AMD x86 CPUs stand as a prime examples for this [Int13, Ch. 10-13].

In the case of software, data parallelism can be exploited by the compiler that, after analysis, parallelizes sequential sections of the program. It can also be annotated by the programmer to assist the compiler generate code for optimal utilization of the systems (e.g. CPUs) functional units.

A data parallel version of Fortran is known as High Performance Fortran. It adds directives (`!HPF$ INDEPENDENT`) and also syntax extensions (`FORALL`) to Fortran [For97, Ch. 5.1]. With OpenMP users can annotate loop constructs for work sharing using control directives (`#pragma omp parallel`) [Ope11, Ch. 2.6.1]. OpenMP is linked to the shared-memory computational model that uses threads to compute inside a shared address space.

2.1.2 Shared-Memory

For shared-memory the interdependence is not as much specified by the data, but by the programmer and can therefore be classified as a form of control parallelism. Shared-memory is mapped into the address space of other processes and can then be operated upon using the CPUs load and store instructions (Figure 2.1). Even though modern

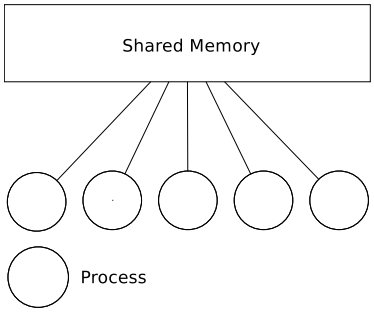

Figure 2.1: The shared-memory model

operating systems, such as Linux, can scale up to thousands of CPU cores, the scalability of shared-memory systems remains limited owing to increasing complexity and coinciding high cost. For example the SGI Altix UV can scale up to 4096 cores with 32 TB of memory using SGI NUMALink 5, a highly specialized cache-coherent Non-Uniform Memory Access (ccNUMA) interconnect [Spi11, STJ+08]. NUMA means that all memory is accessed using a memory transfer network with parts of the memory residing local to each processor or set of cores and therefore differing access time. That is also the reason why it remains unclear, if NUMA systems can be considered "true" shared-memory systems as NUMA implies, memory references are not accessed in a uniform manner.

2.1.3 Message-Passing

Processes that only have local memory can communicate with other processes by sending and receiving messages. This is called message-passing. In this computational model, each process has to actively take part in the communication by either sending or receiving data (Figure 2.2). This is also referred to as two-sided operations. Two-sided operations lead to explicitly controlled data transfer and implicit synchronization by blocking at sends and receives [MG02, p. 513-514]. This computational model has several conceptual advantages. It hides heterogeneity of software and hardware, of the bus and network topologies (data locality) behind a uniform interface. Moreover when a programmer wishes to exploit heterogeneity, he is not restricted. By not exhibiting direct access to other processes memory, it is also not possible to run into hard to debug distributed memory access violations.

Uniform send- and receive operations of message-passing systems can also be traced more easily across entire parallel computing networks. A profiling (or tracing) tool could for example replace message-passing operations with corresponding profiling versions (e.g. `PMPI_` [Mes09, Ch. 14]). The Zentrum für Informationsdienste und Hochleistungsrechnen (ZIH) at TU Dresden is involved in the development of VampirTrace, a tool that can also use the MPI profiling interface [Jur06].

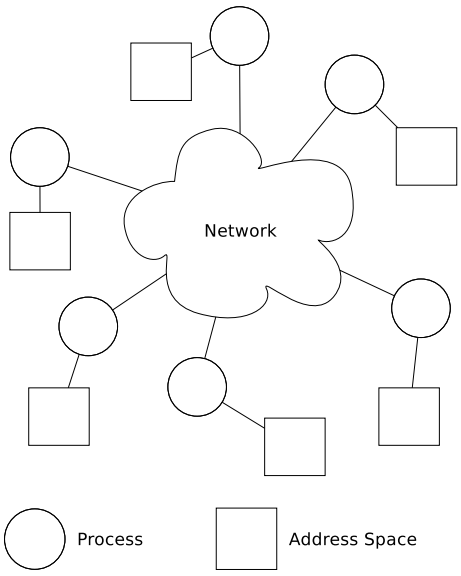

Figure 2.2: The message-passing model

2.1.4 Remote Memory Access

If data transfer and synchronization are separate steps, e.g. both are explicit, the communication operations are referred to as one-sided operations. This allows performance improvements when performing multiple data transfer operations protected by only one synchronization operation, thus reducing total overhead [GT07]. Conceptually, Remote Memory Access (RMA) is not message-passing. For data transfer, the origin process has to access memory of the target process. This is achieved using `GET` and `PUT` operations. There are two different synchronization categories in RMA: Active and passive target synchronization [Mes09, Ch. 11.4]. The active target process takes measures to ensure correct ordering of events by calling synchronization operations whereas the passive target does not. The origin process operates on windows that were exposed by the target. Active target synchronization enables fine-grained control over the exposure epoch by both sides including the possibility of overlapping operations (weak synchronization). Active target synchronization is suitable for pairwise communication. If locking only takes place when the window is accessed by the target, multiple origin processes can not enforce mutual exclusion. Contrary to that, with passive target synchronization lock and unlock operations are issued by origin processes on the window exposed by the target process. Hence passive target synchronization can guarantee mutual exclusion of concurrent origin processes and resembles thereby communication similar to the shared-memory model.

2.1.5 Combined Models

Parallel computational models can be combined. For example the Message Passing Interface (MPI) can hide shared-memory communication behind message-passing semantics.

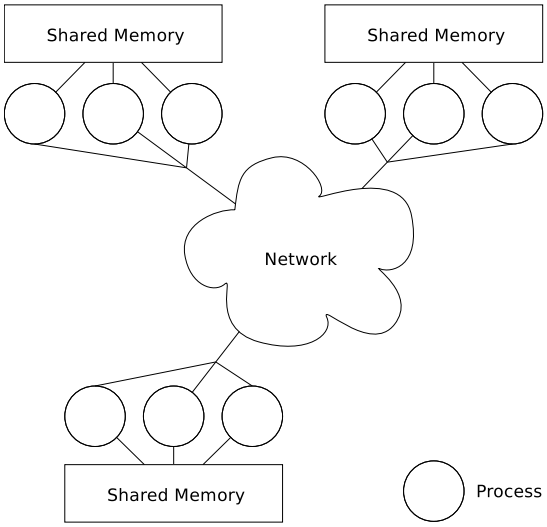

Figure 2.3: The cluster model

Moreover it includes RMA semantics and can be combined with threads or OpenMP into what is called the hybrid programming model. In the cluster model (Figure 2.3) shared-memory systems (Section 2.1.2) are connected by a communication network. As of this writing this is the dominant topology at the top of the performance spectrum.

2.2 Message Passing Interface

MPI (Message-Passing Interface) is a *message-passing library interface specification.*" [Mes09, Ch. 1.1]. In the beginning of the 1990s, a great variety of message-passing platforms for high performance computing existed. The message-passing model had already proofed its usefulness for high performance computing. Software and hardware vendors as well as research institutions were all too often reinventing the wheel building proprietary or somewhat 'incomplete' message-passing platforms. It was in 1992 when at the *Workshop on Standards for Message-Passing in a Distributed Memory Environment* [Wal92] the developers of the most widespread message-passing platforms met to identify essential features for a standard message-passing interface. Only one year later this effort, led by the newly founded MPI Forum, resulted in the release of the MPI-1 specification. This first version of the standard consisted of mechanisms for point-to-point communication (send/receive), collective communication (broadcast, scatter, reduce...), user defined data types; groups, contexts and communicators to ease writing portable parallel libraries, process topologies to aid mapping of processes to hardware benefiting locality, environment management (mainly for error handling and MPI-application attributes), a profiling interface and language bindings for C and Fortran.

One year after the initial release several MPI implementations, MPICH of Argonne National Laboratories and LAM/MPI from Ohio State University among them, followed.

Since then several corrections and clarifications made their way into the MPI-1.3 standard. The next major extension to MPI came with the release of the MPI-2 standard in 1998. It incorporated new features, most noteworthy parallel I/O, remote memory access, extensions to collective operations (non-blocking and all-to-all) and process management. The current release of MPI is MPI-3.0. It was released on September 21, 2012. Most noteworthy it extends the previous standard MPI-2.2 with non-blocking collective operations, more one-sided (RMA) operations and Fortran 2008 bindings.

2.3 MPICH

MPICH (Message Passing Interface Chamaeleon, with Chamaeleon being one of the predecessor message-passing systems by Argonne) is a free and portable (Unix/Windows) implementation of the MPI specification. It has now reached compliance with MPI-2.2 and was the first project to fully implement MPI-1 and MPI-2. The MPICH distribution contains the MPI library, several communication subsystems, test-code, process managers, that start MPI applications, and the multi-processing environment tools for performance analysis, logging libraries, and graphic output. Many software and hardware vendors such as Intel, IBM, Cray, or Microsoft have derived optimized versions of MPICH.

2.4 Process Managers

To start MPI applications, it is necessary to setup a running environment, start processes on a specified processor of a node and to setup basic control channels among the process manager and the spawned processes. This is handled by process managers. MPICH uses the Process Management Interface (PMI) as a wire-protocol [GD12]. It enables processes, that were started by the process manager, to send control messages in a standardized way. In a POSIX environment the process manager will inherit unidirectional pipes and a bidirectional PMI command socket into the newly `fork()`ed process. The unidirectional pipes redirect `stdout` and `stderr` to the process manager for labeling (`rank@node> my msg`). The PMI command descriptor is then either made known using the `PMI_FD` environment variable or acquired by connecting to a `hostname:port`-tuple passed in the `PMI_PORT` environment variable. PMI control commands include set/get for a key value store, barrier synchronization, dynamic process management, general information about the processes (e.g. rank or number of processes) and the initialize/finalize commands.

2.5 The L4 Microkernel Family

2.5.1 Microkernels

A microkernel is an operating system kernel that limits its implementation to inevitable functionality [Lie96b]. The user-mode of a microkernel-based OS therefore comprises of servers that provide all needed functionality, such as device drivers, file systems and

even memory paging [YTR+87, Ch. 3.4]. The user-mode servers communicate with each other using mechanisms provided by the microkernel. This system architecture allows not only for separation of concerns by the inherent modular design but also for strict isolation among servers and applications using virtual memory. Moreover removing responsibility for certain resources from the kernel decreases its code size (trusted computing base) and complexity. Whereas system services in a monolithic kernel communicate by calling functions in the same address-space, microkernel services have to perform inter-process communication (IPC) to call a function of another service in another address-space. IPC via the kernel interface does not only have direct costs for trapping into the kernel, but can also result in costly context-switches. Changing the execution context of the processor can result in a flushed[1] translation look-aside buffer (TLB), that caches resolved virtual-memory mappings and an increased possibility for displacements in data and instruction caches.

2.5.2 L4

The L4 microkernel was originally created by Jochen Liedtke and is the successor of the L3 microkernel. L3 proved that microkernels can be fast, if performance critical aspects are optimized. In practice improving performance means that IPC among microkernel servers must either be avoided or fast. The L3 microkernel showed that IPC can be heavily optimized to achieve performance improvements by a factor of 22 compared with the Mach microkernel. An overview of all the techniques used for implementing L3 IPC can be found in [Lie93]. L4 and L3 both retained scheduling in the kernel for performance reasons. Virtual memory management is separated between the kernel (for CPU exception handling) and a user-level pager that maps pages according to the policy implemented by the region manager. All kernel IPC is synchronous. The original L4 application programmers interface (API) knew only 7 syscalls in total [Lie96a]:

1. IPC for data transfer, interrupt handling, timeouts and mapping or granting pages (flexpages)
2. unmapping of flexpages from other tasks
3. task creation
4. thread creation and execution
5. thread scheduler to set the priority, time-slice or preempter of a thread
6. thread switching for yielding the calling thread or running another
7. id_nearest returns the first IPC peer on the path to destination. That was part of the Clans & Chiefs ([Lie92]) security model

[1] Hardware vendors such as ARM, Intel and AMD have recently introduced tagged-TLBs on their CPU-platforms. Their TLB entries carry an address space tag and do not require flushing of the TLB when an address space is switched.

Today various specifications of L4 derived APIs are available. Important implementations of L4 are L4/Fiasco of TU Dresden, L4Ka::Pistachio of Karlsruher Institut für Technologie, seL4 [KEH+09], the first formally verified microkernel of NICTA (National Informations and Communication Technology Australia Ltd) and OKL4 [HL10] of Open Kernel Labs.

2.5.3 Fiasco

Fiasco, in its current version named Fiasco.OC (Fiasco Object Capability) is a descendant of L4. This kernel, written in C, C++, and Assembly, introduced an object-oriented API where kernel objects correspond to user-level objects. The only remaining syscall is `invoke_object`. Detailed explanations of Fiasco.OC and L4Re, that this section and Section 2.6 are based on, can be found in [LW09]. Five main kernel objects exist:

1. Task - Implements a protection domain consisting of a virtual memory address space and a capability table.
2. Thread - Is a unit of execution. Multiple threads can be associated with one task.
3. IPC-Gate - Is a kernel object without any specific interface. They are used to establish secure communication channels among user-level objects.
4. Factory - Can create any kind of kernel object including factories.
5. IRQ - Sender of asynchronous IPC. IRQs originate from software or hardware. They are bound to a specific thread.

Other kernel objects are the Scheduler (set priority, CPU or timeslice), Icu (access to the Interrupt controller) and Vlog that implements a rudimentary console. All capabilities can be mapped or granted into tasks as well as unmapped from tasks. There is a one-to-one relationship among kernel objects and capabilities. A task's capability table is maintained by the kernel. Therefore indexes to this table are task local references to kernel objects. IPC-Gates are usually created by the implementor of a user-level object and therefore designate the receiver. Upon invoking the user-level object in the sending client, the receiver is automatically mapped a reply capability. The receiver can identify the IPC-Gate, and by that the sender, using a label set at IPC-Gate creation.

2.6 L4Re

The L4 Runtime Environment (L4Re) is the user-land for the Fiasco.OC kernel. Together L4Re and Fiasco.OC constitute an object-capability system. Every IPC-Gate belongs to a user-level object. L4Re has four main user-level objects: **Dataspaces** are abstract memory (e.g. RAM, disk, memory-mapped I/O regions) and can also be shared among tasks (e.g. shared-memory). The **Memory Allocator** makes anonymous memory available using data-spaces. A **Region Manager** handles the address space of a task. This is achieved by assigning a Dataspace to virtual memory and thereby making it visible to the task. It is implemented in the task using a map of

`(memory region, data-space manager)` tuples. When a page-fault occurs, the kernel issues a page-fault IPC to the Dataspace Manager (e.g. a user-level pager) that is responsible for this virtual memory region. **Namespace** objects are used to associate names with capabilities. Namespaces can be nested and are used to implement access control mechanisms. Access to user-level objects is announced by linking user-level object capabilities into the Namespace of the application. Another advantage of this approach is the possibility to substitute user-level objects that share a compatible interface without any changes to the application.

2.7 Related Work

2.7.1 The PARAS Microkernel

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

2.7.2 IBM Blue Gene/L/P/Q - The Compute Node Kernel (CNK)

Blue Gene is a modular supercomputing platform that can scale up to 20 petaflops[2]. Common to all Blue Gene systems is the architecture that subdivides nodes by function. Similar to the PARAM 9000 approach, there are compute nodes and I/O nodes. Both node types can be grouped into variably sized sets of nodes, also known as blocks, partitions or partition sets (psets). The hardware ensures electric isolation of the communication network among the configurable blocks. The Blue Gene series uses a torus network for peer-to-peer communication, a collective network, and an interrupt network for fast barriers [BKM+10, Ch. I]. While I/O nodes run Linux, compute nodes are operated by a custom operating system kernel, the Compute Node Kernel (CNK). This kernel is extremely simple (5000 lines of code) and can not run more then one process.

[2] BG/Q system Sequoia at Lawrence Livermore National Laboratory (LLNL) has 1.57 million processing cores. It achieves a measure peak performance of 16 petaflops, and ranks on position two in the supercomputer top 500 list of November 2012[top].

It is optimized for low OS noise and optimal hardware utilization. At startup, the CNK hands out all resources to the application. The virtual memory area of the CNK is protected. To avoid TLB flushes, that are very[3] expensive on IBM Power PC (PPC) processors, CNK maps all available memory into the application with pages (e.g. 2 GB of physical memory evenly divided into 64 TLB entries yields pages of 32 MB) large enough that no TLB misses can occur. All performance critical communication is performed in user-space to avoid kernel entries/exits and memory copying. OS noise, that can be caused by kernel threads or system services, limits scalability, especially when it is uncoordinated between different nodes. To reduce this all nodes share the same timer source to enable coordinated (noise) events. Nevertheless, compute nodes are never the cause of any OS noise, because the CNK does never reschedule and has therefore only one thread that the kernel is aware of (the hardware thread of the CPU). I/O nodes are also organized in blocks and perform I/O operations for the compute nodes. This is achieved by function shipping and allows to offload latency that could severely reduce scalability and performance and OS noise.

[3] Handling a PPC 440 TLB miss takes $0,27\mu s$ on Linux. Compared to CNK, the default Linux TLB miss handling reduces the NAS Integer Sort benchmark performance run on 32 nodes by a factor of 3[SAB+08, Ch. 6]

3 Design and Implementation

In this chapter I'm going to elaborate design decisions and the implementation that resulted in my MPICH port. The design had to meet the following goals:

1. **Performance** is the most important aspect of message-passing systems. If there are two comparable systems, users will choose the system that offers the better performance.

2. **No or minimal changes** to existing source code. This simplifies updating or migrating the port with future releases of the MPICH distribution.

3. **Don't reinvent the wheel** and use existing, well-tested libraries where applicable.

I will begin the design chapter with process management, because starting it is naturally the first step of running an MPI application.

3.1 Process Manager - gforker

As explained in the previous chapter, process managers start MPI applications. There is a great variety of process managers that support the Process Managment Interface (PMI) of MPICH. The first thing that I had to decide was whether to port an existing Process Manager or to implement a new one.

The default MPICH Process Manager is Hydra. Hydra is a highly complex as well as complete process manager that runs on most POSIX compliant operating systems. This makes porting Hydra to operating systems with limited POSIX support complicated. Therefore I chose to port a more simple process manager. In the following I will illustrate a complete MPI application setup and discuss how each step can be implemented. Following this section I will outline the design of the particular solutions.

Figure 3.1 shows components of an MPI application running on one node using shared-memory for communication. To run an MPI application the user must have a working binary. Compilation and linking it to the MPI library will not be explained here. The user can start his MPI application either as standalone (e.g. running on a single CPU) or with the assistance of a Process Manager. In the first case, neither control channels for process management nor a communication subsystem need to be initialized. If a process manager is invoked, as it is the case in Figure 3.1 (Step 1), the startup procedure gets more interesting. For example when a user executes `mpiexec -n 4 ./mpiapp` from the command line, the process manager (`mpiexec`) will start the MPI application (`./mpiapp`) using four cooperating processes. In this work I will only deal with the nontrivial case involving a process manager.

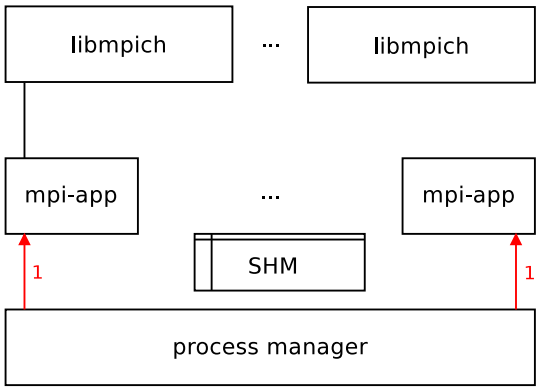

Figure 3.1: Creating an MPI process.

3.1.1 Starting Processes

Creating and launching processes is the first step when setting up an MPI application. In L4Re processes are usually started by a **loader**. The loader prepares the initial environment of an application that mainly consists of environment variables and capabilities, e.g. IPC-gates or namespaces, and launches it. Moreover it is also used to establish communication channels among processes of launched applications by creating one or more communication capabilities, such as IPC-gates, IRQs or shared-memory, and mapping them into the respective task at creation time. Therefore a process manager can be classified as a loader. My solution to launching applications in L4Re is laid out in Section 3.2.

3.1.2 Set Up Control Channels

Installing control channels for an MPI application is straightforward under UNIX. A bidirectional pipe is created by the process manager prior to creating (`fork()`) the process using the `socketpair()` system call. The control channel descriptor is then inherited into the child process. For labeling messages (`stdout` and `stderr`) of MPI processes, the process manager creates unidirectional pipes and inherits the writable descriptor into the child process. Finally the process manager adds the control channel descriptor as well as the standard output descriptors to its set of observed descriptors and waits for events by invoking the `select()` (or `poll()`) syscall in its I/O loop. Porting this functionality to L4Re was the most complicated aspect of this work. A solution to this is described in Section 3.3.

3.1.3 Setup Data Channels

After all processes have been launched and control channels are established (or inherited), each process calls the `MPI_Init()` function of `libmpich`, as shown in Figure 3.2 (Step 2), to begin the initialization. The first initialization step is the establishment of a PMI connection over the control channel. To achieve this, each process needs tp per-

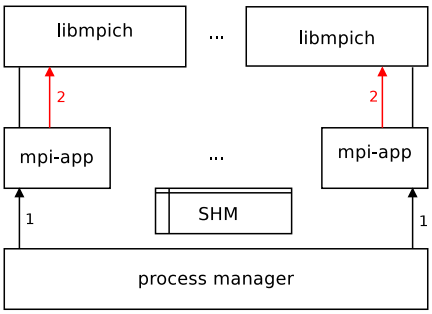

Figure 3.2: Calling MPI_Init() in each MPI process.

form a handshake with the process manager. This inevitably requires to decide: Write or port a process manager?

To **rewrite as standalone** the entire functionality of PMI and mechanisms to launch and connect the processes need to be implemented. This amounts to a lot of work and what makes matters even worse is the prospect that specialized code for launching and doing IPC is unlikely to be reusable. While application specific IPC code is likely to deliver optimal performance, control channels do not require optimized IPC since their sole use is process management and not message-passing of application data.

Using the **hybrid approach** comprising of a separate loader and process manager saves the programmer from writing his own loader and dealing with the details of mapping initial capabilities or setting up the application environment. The loader handles the creation of control channels and the mapping between processes and process manager. PMI, especially handling of the key-value store, can then be implemented by a standalone process management server. To support dynamic process management, the loader needs to provide a server interface that can be mapped as capability into the process manager. As of this writing, ned is the default loader of L4Re and does not provide such an interface. Therefore extending ned adds to the complexity and effort of the hybrid approach.

When **porting an existing process manager**, a decision towards the donating operating environment must be made. There are several system libraries available for Fiasco.OC/L4Re that suggest a POSIX system, such as Linux, is to be preferred over a process manager for Microsoft Windows. The `l4re_vfs` library provides the Virtual File System (VFS), an interface for operations on file descriptors ([SGK+85]). It introduces a generic layer between miscellaneous I/O mechanisms (files, pipes, memory, sockets) and an application. All this file types are abstracted as file descriptors and can be operated upon in a uniform fashion using POSIX functions such as `open()`, `read()` and `write()`.

As mentioned in Section 3.1.2, POSIX process managers perform file descriptors demultiplexing of MPI processes using either `poll()` or `select()`. Currently neither `l4re_vfs` nor other libraries available on Fiasco.OC/L4Re offer this functionality.

Figure 3.3: Summary of alternatives

property/alternative	port	hybrid	rewrite
IPC mechanism			
- performance	⊘	⊘	⊕
- effort	⊖	⊕	⊘
start processes			
- static	⊘	⊕	⊘
- dynamic	⊕	⊖	⊕
avoid POSIX / embrace L4Re	⊖	⊕	⊕
reuse PMI	⊕	⊘	⊖
reuse emerging code	⊕	⊘	⊖

Process startup in most POSIX environments is accomplished by a combination of the `fork()` and `execve()` system calls, that neither exist in the kernel interface nor as a compatibility library in the L4Re userland. As of yet, starting processes without invoking a loader is not common in L4Re. A loading mechanism must also provide a way to inherit file descriptors. Atop of all that, a process creation interface should try to mimic the target interface. That means that creating a process should either be POSIX compliant (`fork()`/`execve()`) or offer an interface that eases porting by exposing an interface taking identical or similar parameters. A ported process manager can be classified as a loader. This simplifies dynamic process management.

Concerning the PMI implementation, the user can rely on mature functionality. Moreover can porting an existing process manager produce artifacts, that can be of use for other ports, including different, possibly more sophisticated, process managers or to write or port a shell. Figure 3.3 summarizes all of the above mentioned views.

In my view porting an existing process manager is the best alternative. Not only the use of well established code but the outlook of creating reusable POSIX-alike interfaces to ease porting other software, including a fully-fledged process manager convinced men. `Gforker` is likely to perform sufficiently well on small to mid scale shared-memory machines. With regard to scalability, huge distributed systems require more sophisticated setup algorithms[1] that can only be found in complex process management frameworks such as Hydra.

3.1.4 PMI and the MPI setup

Having decided on the general approach regarding the process manager, I will now continue with a brief explanation of the PMI setup. Figure 3.4 (Step 3) illustrates a number of PMI calls. Most notable are a handshake to agree on the PMI protocol version (`cmd=init`), the discovering of maximum values (`maxes`) and the name (`my_kvsname`) of

[1] Hydra/PMI2 introduced job attributes that aid the initial setup to initialize an adequate communication channel with regard to topology. This reduces the number of total queries to the key-value store from n^2, where each process queries all other processes, to 0, where the process manager pushes all relevant topology information once. See [BBG+10] for reference.

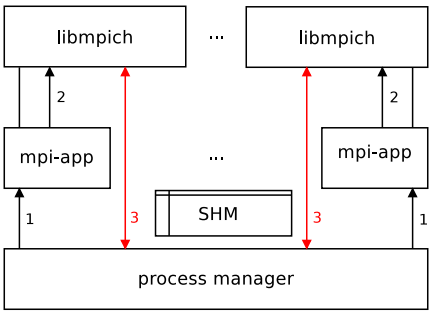

Figure 3.4: Set up PMI, KVS, and discover the other processes.

the key-value store (KVS), communication barriers (`barrier`) to separate KVS put/get requests, and setting/getting the hosts of all processes in the communicator to determine the communication channel (e.g. shared-memory for node-local communication) for each process. A communication subsystem setup for shared-memory is shown in Figure 3.5 (Step 4a). It illustrates the creation of shared-memory and the publishing of a its identifier using a sequence of PMI calls originating from the MPI library (`libmpich`) via the process managers control channel. MPI function calls to underlying devices are translated into ADI3 (Abstract Device Interface 3) calls. ADI3 is very similar to the MPI interface and marks the transition from application (MPI) to device layer. ADI3 issues function calls to devices such as InfiniBand [HSJ+06, Sec. 2.2] or CH3 (Channel 3). CH3 is virtual driver that simplifies adding new communication mechanisms by offering a reduced, more generic interface on top of the more specific ADI3 at low performance cost [BM06, pp. 4].

I chose to port the Nemesis communication subsystem running as a CH3 channel. Nemesis is the current default in MPICH and offers all types of communication (message-passing, RMA) over a great variety of modules for intra- (shared-memory) and inter-node communication (InfiniBand, GigaBit-Ethernet, Myrinet). Moreover [BM06] has shown that the shared-memory module performs well with regard to latency and bandwidth. To setup intra-node communication (shared-memory) among multiple processes, the local group leader (often rank 0) must initialize the communication subsystem. To create shared-memory in a POSIX environment, a node-local file using `open()`/`mmap()` must be created and its size set by `lseek()` and `write()`. After this the shared-memory is mapped into the creators address space and gets published to all other local processes of his group using the KVS.

Even though `l4re_vfs` offers POSIX `mmap()`, it does not support the `MAP_SHARED` flag, hence cannot be used to create named shared-memory. Another thing that makes it complicated to extend `mmap()` with this flag is the availability of filesystems, that are required as a global namespace for shared-memory.

`l4shmc` is a library for shared-memory communication in producer-consumer-scenarios. The idea of this library is to provide chunks of shared-memory that can be uniquely identified across different tasks. Chunks carry a status flag that can be set to either busy, ready for reading/consuming, and clear. By triggering Fiasco.OC

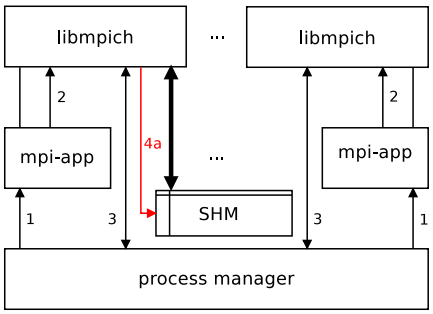

Figure 3.5: Process (local group leader) maps shared-memory and publishes its identifier.

IRQ kernel-objects that are associated with the chunk, consumer and producer can be informed of a chunk status transition.

Because `l4shmc` does neither support deletion of chunks nor growing the shared-memory area, it is not suitable for use as MPICH shared-memory mechanism. I therefore wrote four basic operations to create, attach, detach and delete areas of shared-memory. As in `l4shmc`, I use an L4Re namespace, mapped by the process manager into each of the MPI processes, to make the shared-memory Dataspace capabilites uniquely identifiable. IRQ notification is not necessary in MPICH, because shared-memory access is synchronized entirely using machine dependent instructions[2] such as memory barriers (x86: `{m,s,l}fence`), atomic add/subtract (x86: `lock; {sub,add,inc,dec}`) and compare-and-swap (CAS; x86: `cmpxchg`). The shared-memory operations were integrated into the backend found at `mpi/util/wrappers/mpiu_shm_wrappers.h`[3].

After the group leader has created and attached a shared-memory area, the area must be made known to all other processes of the group that run on the same node. In a POSIX environment this node-wide identifier is a file (e.g. `/tmp/shared_XXXX.mem`). By issuing a

```
put kvsname=kvs_2_0 key=sharedFilename[0] value=l4misc_0
```

PMI command, shared-memory identifiers can be put into a key-value store that is known to all processes that were started by the process manager. Here `l4misc_0` uniquely identifies the shared-memory area. Irrespective of my example all MPI processes may create and publish shared-memory areas.

All processes that were launched by the process manager can query the shared-memory from the KVS with a request of the form:

```
get kvsname=kvs_2_0 key=sharedFilename[0]
```

Following that the processes attach to the shared-memory as illustrated in Figure 3.6 (Step 4b). When the shared-memory setup is complete in all participating processes, the Nemesis communication subsystem takes over, initializes itself, and finishes MPI startup.

[2] MPICH avoids kernel entries and exits (entry + cache) to improve latency.

[3] All shared-memory allocated in this backend must be aligned to cacheline boundaries. MPICH optimizes data structures, memory layout and locking for aligned shared-memory.

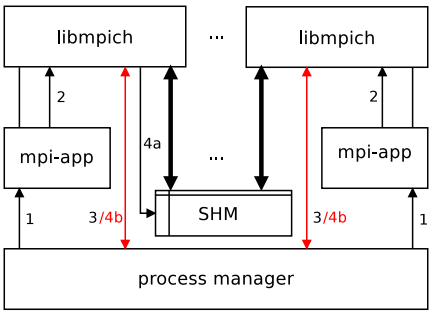

Figure 3.6: Local group processes look up the shared-memory identifier in their KVS and map the corresponding shared-memory into their address space.

3.2 libspawner

As explained in Section 3.1.1 process managers can be classified as loaders. Attributable to the lack of POSIX `fork()/execve()` mechanisms for process creation and binary execution, the need to implement a library mechanism arose. My design objectives were:

1. It must handle parameters of its POSIX counterparts (e.g. `char *argv[]`) to ease porting.
2. It must not reinvent the wheel - we already have a working low-level loading library called `libloader`.
3. It must be easy to use, have sane defaults and protect the user from details.
4. It must be reusable.

I settled for a C++ implementation, not only because the loader library is written in C++, but also because the encapsulation of state in objects benefits design goals 2, 3 and 4. Figure 3.7 shows an overview of public methods in the `libspawner` API. Using this API, starting an application with minimal (default) setup requires low effort and know-how from the user. Code for minimal process creation is shown in Figure 3.8. By default the `Spawner()` adheres to the principle of least authority (POLA; also principle of minimal privilege). POLA is a design pattern for secure systems. It requires the implementation of fail-safe defaults ([SS75, p. 1282; b) + f)]) what "limits the damage that can result from an accident or error". Fail-safe defaults necessitates to explicitly enable access to information, like environment variables and objects, such as capabilities. Explicitly permitting access requires precise specification of all needed components what can be hard or impossible to know in advance for dynamic usage scenarios. Consider for instance a shell. Shells are used to start a wide variety of different applications of which some require more and some less access to information and capabilities. Even though it would be possible to specify exactly what access should be permitted for every single application, this would require a lot of configuration work, and thereby conflict with design goal 3.

Figure 3.7: Libspawner API

```
Spawner:
  Spawner(bool inherit_caps = false, bool inherit_envs = false)
  App_ptr spawn(const char *filename,
                char *const argv[] = NULL,
                char *const envp[] = NULL,
                unsigned cpu = INHERIT_CPU,
                unsigned prio = INHERIT_PRIO)
  int kill(App_ptr& app_ptr)
  int wait_app(App_ptr& app_ptr, Wait_state* ws = NULL,
               int flags = 0)
  int wait(Wait_state* ws = NULL, int flags = 0)
  void set_cap(l4re_env_cap_entry_t& cap_entry)
  void set_cap(char const *const name, l4_cap_idx_t cap,
               l4_umword_t flags = 0)
  void remove_cap(l4re_env_cap_entry_t& cap_entry)
  void remove_cap(const char *name)
  void set_env(char const *env)
  void remove_env(char const *key)
  void set_dead_apps_handler(void (*da_handler)(int))
  void set_flags(l4_umword_t l4re_dbg,
                 l4_umword_t ldr_flags)
```

Figure 3.8: Minimal process creation with libspawner.

```
Spawner s = new Spawner;
s.spawn("rom/app");
```

Process managers and MPI applications share parts of the environment. The environment can be altered by the user to control the MPI application. Imagine a user that wants to start an MPI application. The user configures the loader to map all capabilities and set all environment variables required by the process manager as well as the MPI application. The latter makes implementing fail-safe defaults impossible, because the process manager does not know the capabilities or environment the MPI application requires. In this usage scenario, the expected behavior of the process manager is to pass-through the complete environment as intended by the user when configuring the loader.

To enable passing-through of environment variables and capabilities, I added two parameters to the `Spawner`s constructor. By default the Spawner adheres to the POLA. In case of a more tightly coupled process execution, as it is the case with a process manager and an MPI application, this can be altered to enable a more POSIX-alike behavior that inherits all environment variables and capabilities. To control this, set

and remove methods have been added that allow to reduce and extend the sets of environment variables and capabilities `libspawner` passes to new processes.

3.3 libendpoint

As mentioned in Sections 3.1.2 and 3.1.3, UNIX process managers use bidirectional pipes, as created by the `socketpair()` system call, and demultiplexing mechanisms such as `select()` and `poll()`. To my knowledge, no similar mechanisms are available in the L4Re userland and its libraries. Moreover inheriting file descriptors into created processes without changes to the source of the inheriting process is desirable.

IPC-gates, as explained in Section 2.5.3, provide a way to implement synchronous IPC in the Fiasco userland. Bidirectional pipes require two IPC-gates for every pair of peers. A `poll()` for readable descriptors could be implemented using an open IPC receive (receive-from-all) on the server side. Implementing `poll()` for writable descriptors would be more complicated, since a notification mechanism that indicates whether a `write()` is going to be blocking, would be required.

IPC can also be implemented with shared-memory. Producer-consumer scenarios (pipes) can be implemented by sharing a buffer and its associated status and some form of mutual exclusion. A typical sequence could be: The producer reads status=`CLEAR`, sets status=`BUSY`, writes in the buffer, and sets status=`READY`. Then the consumer reads status=`READY`, reads the buffer and updates the status to `CLEAR`. Often neither producer nor consumer resort to busy waiting when the status flag does not permit access, but block until a wakeup event (ready for reading or finished reading) triggers a status reexamination.

The shared-memory producer-consumer library in L4Re is named `l4shmc`. `libendpoint` is based on this library. As described in Section 3.1.3, this library provides named shared-memory. My implementation employs software IRQs[4] to signal readiness after writes and completion after reads. IRQs offer a fast asynchronous notification mechanism via the Fiasco.OC microkernel. Upon triggering an IRQ by invoking its capability, the thread that is attached to that IRQ kernel-object can receive the IRQ event by an IPC receive. This is sufficiently fast, especially when considering that `libendpoint` is only used for the process manager's control- and output redirection pipes.

Figure 3.9 is a listing of the `libendpoint` API. I defined an interface that can easily be used and extended. In the most cases the user of this library will only need four methods:

- `link()`, to associate two `Endpoint`s,
- `connect()`, called simultaneously to setup the connection (e.g. signals and transport medium) between a pair of linked `Endpoint`s into the context of the calling thread,
- `read()` and

[4] If the necessity to improve bandwidth and latency would arise, a busy waiting version (no kernel entries/exits) can be derived.

- `write()`.

All connections between pairs of `Endpoint`s are bidirectional to simplify use and implementation. `Endpoint::write()` and `Endpoint::read()` can be nonblocking, if the underlying Endpoint implementation supports it.

Figure 3.9: Libendpoint API

```
Endpoint:
  Endpoint()
  unsigned long int label_done_out()
  unsigned long int label_ready_in()
  long wait_in(l4_timeout_t to)
  long wait_out(l4_timeout_t to)
  bool is_readable()
  bool is_writeable()
  long connect(long timeout_ms = 5000)
  size_t write(char const *const bytes, size_t len)
  size_t read(char *const bytes, size_t len)
  long wait_any(l4_timeout_t to, unsigned long int *label)
```

`Shm_endpoint` is a shared-memory implementation of the `Endpoint` interface. The constructor and `link()` prototypes are specific to the implementation and their interface may vary. Calling the constructor with `flag = Non_blocking` configures `read()` and `write()` to be nonblocking. When destroying the `Endpoint`, the destructor attempts to hangup the connection gracefully. I believe that the POSIX `EOF` condition is a good way to terminate a connection and that employing it will aid porting applications that check the return value of `read()` for `EOF`, especially after returning from `poll()`.

An `Ep_group` provides demultiplexing similar to POSIX' `poll()` system call. `Endpoint`s of different implementations may be added to the same `Ep_group`. To wait for state transitions in the set of `Endpoint`s, `Ep_group::poll()` is invoked and blocks until either the state of one or more `Endpoint`s change (e.g. `Endpoint`s get write- or readable or a timeout occurs). Considering fairness and especially to avoid starvation, `Ep_group::poll()` is arbitrating by a round-robin scheme. I.e. upon invocation, `Ep_group` continues with the next `Endpoint` in the list and starts over after reaching its end. There are several ways to implement demultiplexing under Fiasco.OC that each have advantages and disadvantages. The POSIX interface allows to change the set of file descriptors with every call. This was not necessary to port the process manager and therefore not implemented. `Ep_group` is performing an open receive in the process managers main thread. This is sufficient for the process manager, since no other IPC is taking place in this thread. To implement POSIX-style `select()`/`poll()` semantics with `libendpoint`, every `Endpoint` needs to receive Irq-IPC in its own thread and trigger an IRQ for for all `Ep_group::poll()`-threads it is associated with (if any) when an event occurs. This would increase the overhead compared with my simple implementation, but enables clean isolation of IPC events. More importantly it

avoids expensive IRQ attach/detach operations (2000 ops for 1000 file descriptors) with each call to `Ep_group::poll()`, because association of an `Endpoint` with a call to `Ep_group::poll()` can be done in user-space.

To make use of `libendpoint` when porting applications, there needs to be some sort of POSIX integration. To get file descriptor inheritance working, I had to integrate `libendpoint` into `l4re_vfs` and automatically startup, i.e. `connect()`, the inherited `Endpoint` into a specified file descriptor of the new process.

`Endpoint` to `l4re_vfs` integration is straightforward. All `l4re_vfs` file descriptors are backed by an implementation of the `Be_file_stream` interface. For a working `Endpoint_stream` only methods `readv()` and `writev()` were required and implemented.

To facilitate automatic setup, a library constructor was added that, if the library is a dynamic shared object (DSO), does not need explicit invocation. An initial capability gets evaluated for automatic setup when its name is of the form `autoep_<fd>`, with `<fd>` being a placeholder for the file descriptor, including standard in- and output, that the `Endpoint` shall be installed into. Additionally the setup routine performs two sanity checks. Those are comparing the `Endpoint`'s configuration capability (a mapped `L4Re::Dataspace`), to match a certain magic key and expose a supported `Endpoint` type (e.g. `Shm`). Finally an `Endpoint` object according to the type is created and its `connect()` method called.

3.4 Miscellaneous functionality

In this section I give a brief overview of all the *glue code* that was used to make MPICH and the process manager work. This glue code addresses two necessities:

1. C++ Libaries (`L4Re`, `libendpoint` and `libspawner`) need to be wrapped into C functions.
2. Avoid code changes and therefore implement application specific POSIX/POSIX-alike functions.

The functions listed in Figure 3.10 were necessary to implement the process manager.

POSIX `poll()` takes an array of a structure composed of file descriptors and their respective requested as well as returned events. This array may change with every single call. Implementing this is either extremely slow (IRQ attach()/detach()) or a lot of work (thread(s) attaching to the IRQs; blocking of `poll()` is intermingled with blocking of `read()` and `write()`). `libendpoint` is simpler in this respect and adds `Endpoint`s (file descriptors) only once. `l4misc_spawnve()` starts a process, establishes three `Shm_endpoint` channels (`stdin/stdout`, `stderr` and `PMI`) into `l4re_vfs` file descriptors, and returns a process id. With `l4misc_sigchld()` a function can be registered as a `SIGCHLD` handler. Its implementation does not interrupt the main thread but runs a second thread concurrently. All `POSIX` wrappers are incomplete but sufficient implementations for the process manager port.

The following functions are necessary for the MPICH library. `l4misc_init()` and `l4misc_finalize()` are called by `MPI_Init()`/`MPI_finalize()` operations to perform

Figure 3.10: Miscellaneous functions and POSIX bindings used to port gforker

```
int  l4misc_spawnve(const char *filename, char *const argv[],
                    char *const envp[], int *err,
                    int *inout, int *pmi);
/* wrapped into POSIX signal() */
void l4misc_sigchld(void (*sc_handler)(int));
int  l4misc_poll(int *fd, int timeout_ms);
void l4misc_poll_add(int fd, long flags);
void l4misc_poll_remove(int fd);

/* wrapped into POSIX kill() */
int l4misc_kill(pid_t pid, int sig);
/* wrapped into POSIX wait() */
int l4misc_wait(int *status);
/* wrapped into POSIX waitpid() */
int l4misc_waitpid(pid_t pid, int *status, int options);
```

Figure 3.11: Implementation of named shared-memory for the MPICH library

```
void l4misc_shm_create(l4misc_shm_t **shm, size_t size);
void l4misc_shm_attach(l4misc_shm_t **shm,
                       const char *shm_name);
void l4misc_shm_destroy(l4misc_shm_t *shm);
void l4misc_shm_detach(l4misc_shm_t *shm);

void l4misc_init(void);
void l4misc_finalize(void);
```

any[5] initialization steps. When building static MPI applications (no DSO constructors), this function call a routine that sets up `Endpoints` into file descriptors.

The shared-memory API (Figure 3.11) is used by the `MPICH` library. `L4Re::Dataspace` capabilities are created and registered in an `L4Re::Namespace` mapped by all MPI processes. All shared-memory acquired from this interface is eagerly mapped. The identifier of the shared-memory is unique.

[5] They are also the right place for *mallopt()* settings when measuring.

4 Evaluation

In the following section this port is evaluated with regard to performance in a shared-memory setup. This will be done by executing several benchmarks for one- and two-sided operations.

4.1 Experimental Setup

4.1.1 Hardware

A Dell T7500 Precision was used for all tests. Its configuration consists of two Intel XEON X5650 CPUs. Each socket has 6 cores with each core having a dedicated L1 cache of 32kB and an L2 cache of 256kB. The 6 cores of one socket share a common L3 cache of 12MB. Both sockets communicate using the Intel QuickPath Interconnect (QPI).. I disabled the following features in the BIOS for the measurements:

1. NUMA - Non uniform memory access, that would divide the main memory into two separate partitions.
2. SMT - Symmetric Multithreading (aka. HyperThreading) because I wanted to ensure a one-to-one relation between every CPU thread to any CPU core.
3. Intel TurboBoost that, depending on the core utilization, overclocks the CPUs (2.66MHz->3.06MHz) because it could result in irreproducible measurements.

All test images were loaded using the tftp protocol supported by the network adapter. In- and output were provided by a serial console.

4.1.2 Operating System / Test environment

4.1.2.1 Linux

The base Linux system was provided by Ubuntu Desktop 12.04.1 LTS 32 bit distribution. To enable 12 cores on a 32 Bit system, the kernel had to be recompiled with options `BIGSMP` and `NR_CPUS=12` set. The stable Linux kernel of version 3.7.6 was used for this. Before any measurement on Linux, I reduced OS noise by disabling unnecessarily running services (mainly the desktop) and started measuring when I reached 15-20 wakeups/second (measured with Intel powertop; 8 wakeups/second for rcu). All Linux tests were run from `ramfs` to avoid any noise that could be introduced by the kernel flushing writes to disk (logfiles).

4.1.2.2 Fiasco.OC/L4Re

Fiasco.OC was compiled with support for multiprocessing and with `MAX_CPUS=12`. The fixed priority scheduler was used. Every thread that is started must be explicitly assigned to a specific CPU because his scheduler does not implement any policy with regard to CPU affinity or thread migration.

4.1.3 Compiler

`GCC 4.6.3`, the standard compiler of Ubuntu 12.04.1 LTS was used for the compilation of binaries for both platforms. I ensured that all performance critical compiler flags[1] were identical when building the benchmarks.

4.1.4 Libraries

A important difference between Ubuntu Linux 12.04.1 and Fiasco.OC/L4Re comes from the different userland. While Ubuntu is employing `glibc` 2.13-1, L4Re uses `uclibc` 0.9.27. While the development of `glibc` is targeted mainly towards completeness and performance, especially on desktops and servers, `uclibc` tries to implement the complete feature set of a C library targeting minimal binary size, that is needed on embedded platforms. Core libraries, such as `libc` or `pthreads` can introduce performance differences on different platforms. Base libraries aside, I compiled and configured all libraries using identical flags[2].

4.1.5 Timing

MPICH for Linux as well as Fiasco.OC/L4Re use the x86 `rdtsc` (read time stamp counter) instruction to measure time. The benchmarks invoke `MPI_Wtime()` twice for every iteration. The time difference can be calculated with:

$$measured\_time := end\_time - start\_time$$

The `rdtsc` instruction stores a 64 bit value containing the count of clock cycles since the last processor reset. To be able to map the measured time stamp into seconds, the MPI library measures the CPU clock by invoking `gettimeofday()` as well as `rdtsc` twice with a `usleep(250000)` between both calls. The formula

$$MPID\_Seconds\_per\_tick := \frac{gettimeofday\_end - gettimeofday\_start}{rdtsc\_end - rdtsc\_start}$$

[1] Binaries on both systems were built using the following gcc flags: *-static -fomit-frame-pointer -g -O2 -fno-strict-aliasing -fno-common -std=gnu99 -m32 -march=pentium4 -fno-stack-protector*

[2] MPICH was configured using *./configure –prefix=<path> –with-pm=gforker –enable-threads=runtime –enable-g=none –enable-fast=nochkmsg,notiming,ndebug –disable-f77 –disable-fc –disable-cxx –enable-smpcoll –enable-timer-type=linux86_cycle*. Option *linux86_cycle* enables the *rdtsc* instruction for MPI timing directives and uses *gettimeofday()* to discover a seconds-per-cycle ratio. *nochkmsg* disables additional error checking while *notiming* disables time measurements in functions. *-enable-threads=runtime* lets the user dynamically decide if locking takes place when entering MPI functions

calculates the period of the CPU and thereby the clock by:

$$clock := \frac{1}{period}$$

This is very accurate for Linux with a measured clock ranging between $2.65999 - 2.66001 GHz$ ($0.000000000375939 - 0.000000000375941 s/clockcycle$). Fiasco.OC offers two options how the clock field in the kip (kernel info page) should be updated. The first method makes use of the scheduler granularity, the second acquires timestamps with `rdtsc` instructions. Using the scheduling granularity resulted in a measured clock speed of $2.621 GHz$. For some runs, I also measured clock speeds of up to $5.51928 GHz$. The reasons for these outliers are unknown to me. Using `rdtsc` improved the precision of the measured clock speed and resulted in values ranging $2.65983 - 2.67043 GHz$. To improve the precision of my measurements on Fiasco.OC/L4Re, I statically assigned $MPID\_Seconds\_per\_tick := 0.000000000375940$ ($= 2.66 GHz$).

4.2 Tests and Benchmarking

4.2.1 mpich-test-2012

To check the correctness of the MPICH port **mpich-test-2012** was run. It comprises of 145 tests (binaries) that can be run with any MPI implementation. The black-box method tests that a selected function, using a specific input vector, results in an acceptable output vector according to the MPI specification. All tests in **mpich-test-2012** are black-box tests. The tests covered point-to-point communication, collective, environment, context as well as topology operations. All tests contained in this set of tests passed successfully. There are other tests that I have not run because of limited time (porting and running the test suite) and that are likely to test either other operations or differently.

4.2.2 skampi and perftest

4.2.2.1 skampi

`skampi` is the most recent (release 5.0.4, April 2008) of the three benchmarks. Users can configure their own set of benchmarks into a `skampi` input file. The language used for describing tests supports calls to MPI-test primitives, for-loops and several configuration parameters such as minimum/maximum repetitions of a test, maximum tolerable relative standard error and buffer sizes[3].

[3] set_min_repetitions(16); set_max_repetitions(32); set_max_relative_standard_error(0.03); switch_buffer_cycling_on(); /* Reduces cache effects. */ set_send_buffer_alignment(64); set_recv_buffer_alignment(64); set_cache_size(24MB); /* Deactivated in the skampi source. */; set_skampi_buffer(4MB); /* Defines the maximum message size; different for different tests */

4.2.2.2 perftest

`perftest` by William Gropp and Ewing Lusk is a set of MPI benchmarking tools for MPI-2 communication directives. The paper *Reproducible Measurements of MPI Performance Characteristics* [GL99] by the two authors contains a list of important aspects of benchmarking and explains how `perftest` tries to adhere to them. In a nutshell, `perftest` runs two loops around each specific measurement. The inner loop controls the message size and the outer loop controls how often each message size needs to be repeated. Measuring different sizes sequentially eliminates the impact of irreproducible effects, such as OS noise, that can last for more then one iteration of the outer loop. This method is based on the assumption, that OS noise is uniformly distributed in time. At the end of the test, *only the minimum execution time* achieved for a certain message size is used. The paper argues that, the minimum measured time is the only reproducible value for the performance of a benchmarked *communication directive*.

4.2.3 Limitations

4.2.3.1 Placement

The first limitation of is the lack of a placement routines in the ported process manager. Sophisticated process managers gather, advertise and apply topology information of the system. For example the `Hydra` process manager places processes based on topology information gathered by `libhw`. This library either reads a topology description from XML, requests it by specific system calls (Linux) or discovers it by measuring CPU-to-cache, cache-to-cache and cache-to-memory dependencies[4]. Figures 4.1 and 4.2 exemplify the influence of placement on measured times for an `MPI_Put()`[5] benchmark run repeatedly with random affinity.

This yields two diagrams that contain placement related performance corridors that look similar on both platforms. To underline that this is the result of placement, I ran this benchmark with static placement (CPU affinity was CPU[rank]) on Linux (Figure 4.3).

[4] Which CPUs share caches? What is the size of caches? How are sockets connected?

[5] The benchmark performs RMA Puts with two groups of four processes. The *-putoption fence* enables several assertions for *MPI_Win_fence()*. Essentially, it makes the implementation aware that no RMA operations precede the first and succeed the second fence. Moreover it notifies the implementation that no stores were performed on the local window within the RMA epoch.

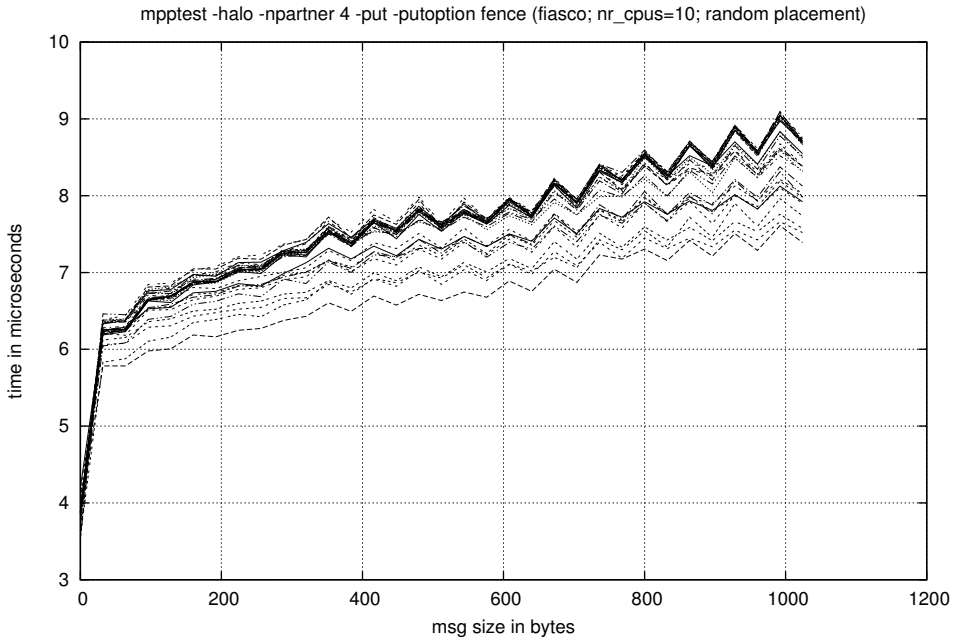

Figure 4.1: Fiasco RMA MPI_Put() with 4 partners; 10 MPI processes (2 groups of 4, 2 idlers). No affinity set.

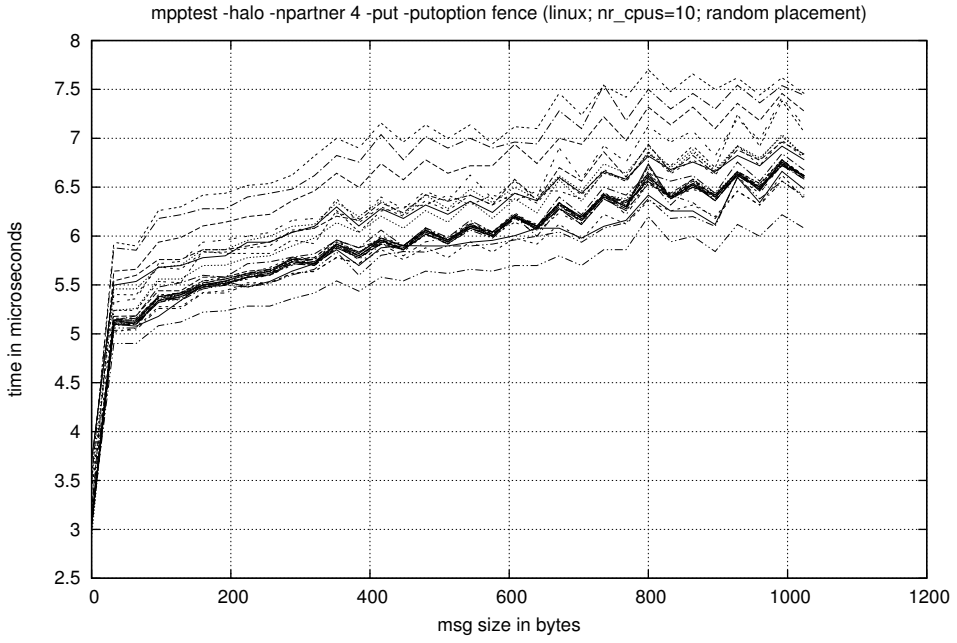

Figure 4.2: Linux RMA MPI_Put() with 4 partners; 10 MPI processes (2 groups of 4, 2 idlers). No affinity set.

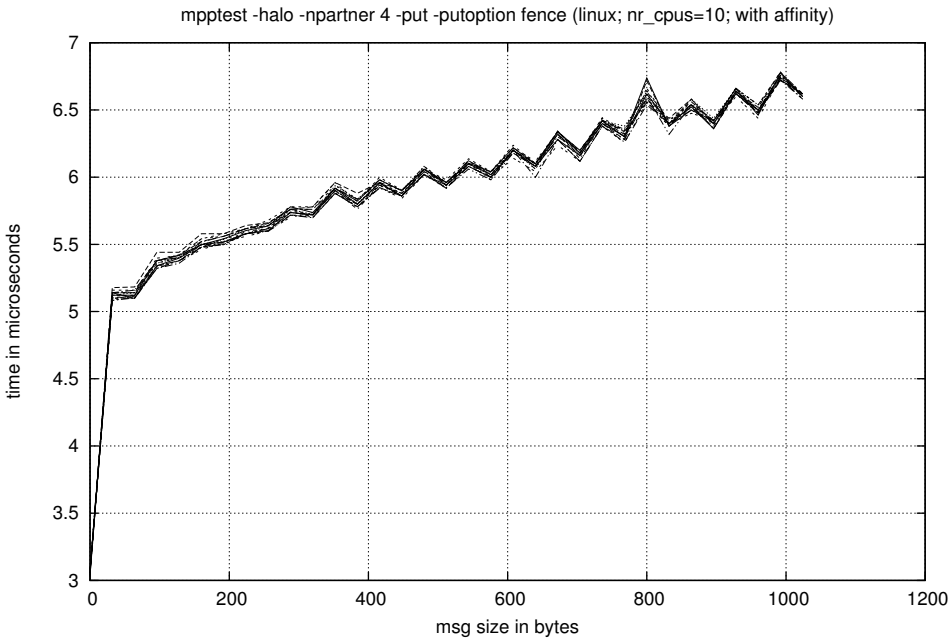

Figure 4.3: Linux RMA MPI_Put() with 4 partners; 10 MPI processes (2 groups of 4, 2 idlers). Affinity set.

Figure 4.3 shows that the benchmark produces reproducible, i.e. almost identical, results when benchmarking conditions (placement) remain unchanged.

I discuss the performance differences between Linux and Fiasco, that are conjecturable from Figures 4.1 and 4.2, in Section 4.3.2.

4.2.3.2 Memory Management

The second limitation is related to memory management, i.e. paging. When a virtual address is referenced and the virtual page containing this address has not been referenced before, a physical frame (e.g. RAM) of the size of a page is assigned to this virtual page. This is known as demand page fetching. The first reason for this paging strategy is to improve responsiveness by not having to page in the whole memory at once. The second is that applications often allocate more memory then required to complete their work resulting in some pages never being accessed. In this situation, physical memory is wasted. In Fiasco.OC/L4Re, paging is implemented by user-level pagers. This frees the microkernel from paging policy code at the expense of performance.

Page faults usually occur when touching memory for the first time. Most MPI applications are memory intensive. When starting as well as during execution, huge buffers are continuously allocated and freed. Running an MPI application results in similar memory allocation and initialization behavior among MPI processes, i.e. local and shared memory of common proportions gets created and mapped. This results in MPI processes competing for the `pager` (`moe`). Figures 4.4 and 4.5 show various runs

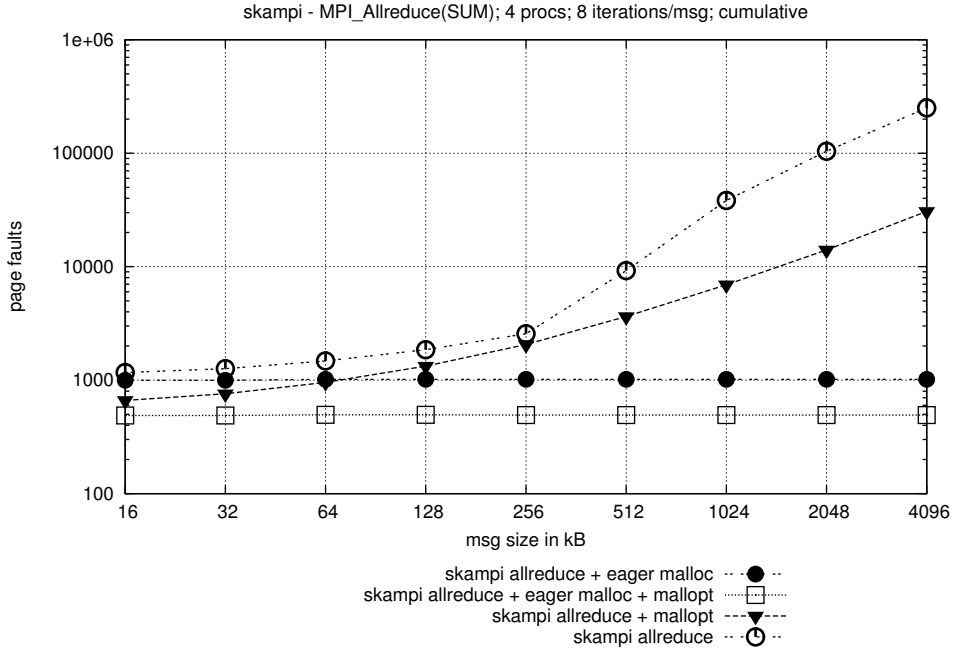

Figure 4.4: MPI_Allreduce(): Maximum message size to page faults. Repeatedly run with alternative maximum message size.

of the `MPI_Allreduce` benchmark. It illustrates several configurations of `malloc()` and their impact on the amount of page-faults (Figure 4.4) and attach/detach region operations (Figure 4.5) with differing sizes for the maximum message buffer.

When eager_mapping is set, all memory acquired from `malloc()` is already paged in (present). `mallopt()` disables the use of `mmap()` for large (`M_MMAP_THRESHOLD`) memory requests and serves only memory allocated on the heap[6]. There are three areas where MPI applications allocate huge amounts of memory:

1. In the shared-memory subsystem (including shared-memory RMA windows)
2. With `malloc()`-type functions on the heap.
3. Using `mmap()`-type functions, including `malloc()` for huge[7] memory allocations.

As for shared-memory, I use eager mapping what improved the situation regarding page-faults. With more memory allocation on the heap or by `mmap()`ed memory, the

[6] In L4Re the "heap" is *mmap()*ed memory. Throughout this work, I use the term heap as synonym for memory acquired from *malloc()*, that was not allocated to a dedicated region mapping. The *mallopt()*s in use were: *mallopt(M_MMAP_MAX, 0)* to disable *mmap()*ing outside of the heap, *mallopt(M_TRIM_THRESHOLD, 8MB)* to shrink the heap when the unused memory at its top exceeds 8MB and *mallopt(M_TOP_PAD, 1MB)* to have a generous slack space.

[7] *malloc()* serves memory requests that exceed 128-512kB (a typical default value for *M_MMAP_THRESHOLD*) using *mmap()*. This guarantees that the entire memory can be reclaimed when *free()* is called and thereby limits fragmentation of the heap.

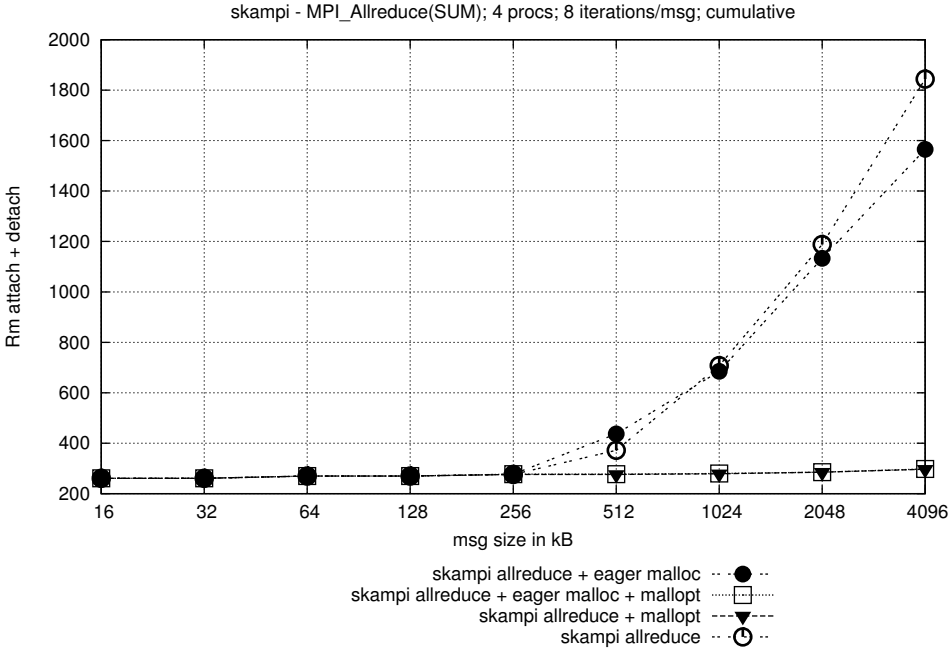

Figure 4.5: MPI_Allreduce(): Maximum message size to region attach+detach. Repeatedly run with alternative maximum message size.

importance of shared-memory page-faults diminishes. A quick (and dirty) solution to 2. provides eager mapping of all `mmap()` requests to anonymous memory (`MAP_ANONYMOUS`). This works because `uclibc`'s `malloc()` and `free()` use `mmap()` to grow the heap. While this does reduce page-faults, there are MPI operations, such as `MPI_Allreduce()` and `MPI_Sendrecv_replace()`, that allocate (potentially huge) temporary buffers. As mentioned above, `malloc()` uses `mmap()` if the requested allocation exceeds a certain threshold. After completing work and before returning from the MPI operation, this buffers are `free()`d. This means that for every invocation of such an MPI operation, memory is `mmap()`ed and `munmap()`ed causing high overhead, even when memory is eagerly mapped. Benchmarking of `MPI_Allreduce()` follows in Section 4.3.1.2.

While eager mapping can help to reduce the overhead, it remains high, especially when compared with Linux.

4.3 Benchmarking Results

4.3.1 Two-sided Operations

This section contains measurements of various MPI operations. I will briefly explain the semantics of each measured operation and interpret what can be seen in the diagrams and what I think causes the performance differences. The explanations to MPI operations largely follow the MPI specification [Mes09].

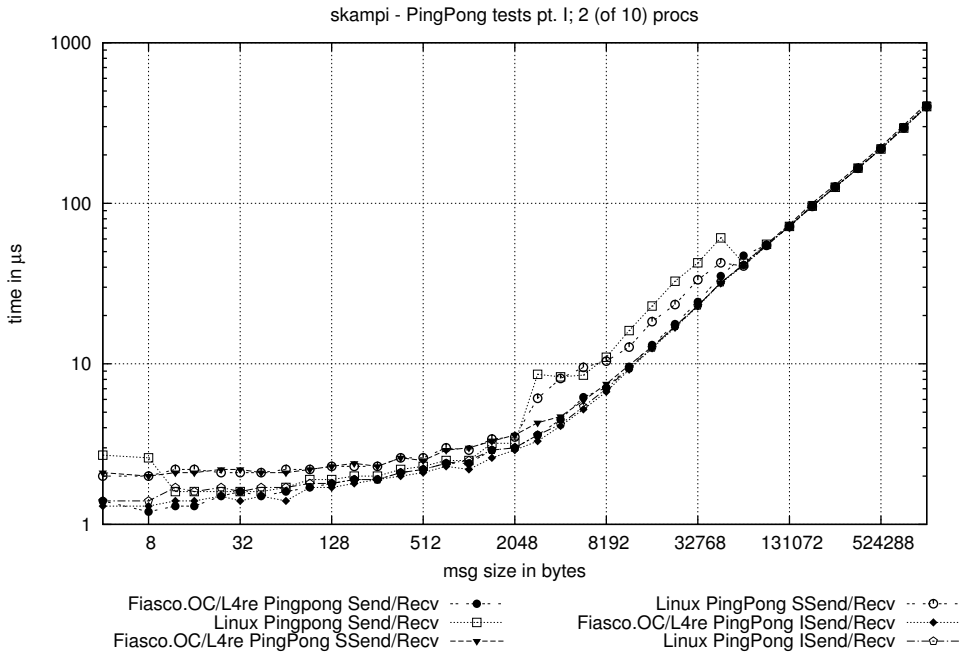

Figure 4.6: Various MPI send/recv operations pt. I

4.3.1.1 Point-to-point Operations

Figure 4.6 `MPI_Send()` performs a blocking send. Blocking means that a return from this function guarantees that the *application's* send buffer is free for reuse. The MPI standard does not guarantee that the message was already flushed from the *system* buffers or received by the destination process. `MPI_Recv()` blocks until the message was received and is present in the application buffer. `MPI_Ssend()` blocks until the destination process has started to receive the message. `MPI_Isend()` performs a nonblocking send. The application buffer may not be altered until `MPI_Wait()` or `MPI_Test()` performed on the request handle set by `MPI_Isend()` indicate that all data has been copied from the application buffer.

Figure 4.7 `MPI_Sendrecv()` does a send and a receive operation (`MPI_Isend()` and `MPI_Irecv()`) operation concurrently and waits (progress engine similarly used by `MPI_Wait()`) until both operations have finished. `MPI_Irecv()` begins a nonblocking receive. The user must ensure that all data is present in the application's receive buffer by calling wait or test operations on the request handle. `MPI_Iprobe()` returns true if a message from a specific source/tag and group is available. `MPI_Iprobe()` is polled in a tight loop by `skampi`. The point-to-point benchmarks were performed only between processes of rank zero and one in the default communicator's (`MPI_COMM_WORLD`) group. All other started processes were idle. There seem to be only little performance differences when running this benchmarks on both platforms.

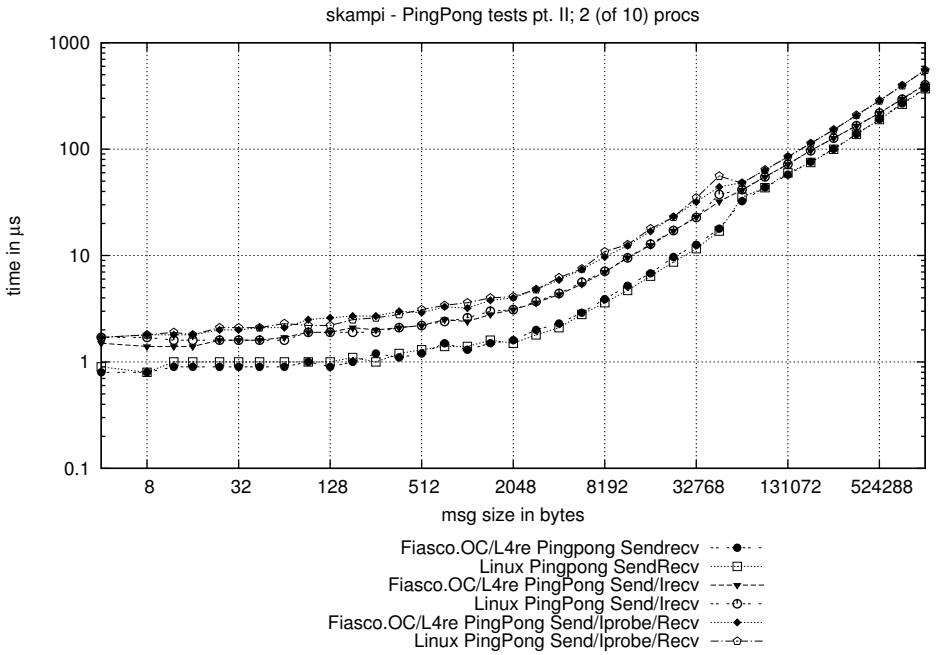

Figure 4.7: Various MPI send/recv operations pt. II

4.3.1.2 Collective Operations

In the previous section point-to-point operations were benchmarked. They are the basic building blocks of collective operations and therefore directly influence their performance.

Figure 4.8 The first collective operation that I want to discuss is the `MPI_Allreduce()` operation. In Section 4.2.3.1 I discussed effects of memory management and page-fault handling on MPI performance in context of this operation. All of the collective operations were measured using 10 cooperating processes to avoid running the pager and an MPI process on the same hardware thread. `MPI_Allreduce()` is a blocking operation. In the first phase it combines the input vectors of all processes in a group using a supplied operation (e.g. minimum, sum, product or any user defined function). In the second phase the result is broadcast into the output vector of each participating process.

The performance of `MPI_Allreduce()` is almost identical for message sizes less than 128kB. As mentioned in section 4.2.3.2, `malloc()` starts allocating memory using `mmap()` when a certain threshold is crossed what hits MPI performance.

Figure 4.9 `MPI_Bcast()` is a blocking operation that returns after the buffer supplied by the *root* process has been copied into the the buffer of every other participating process. `MPI_Reduce()` has the same semantics as `MPI_Allreduce` except that it does not

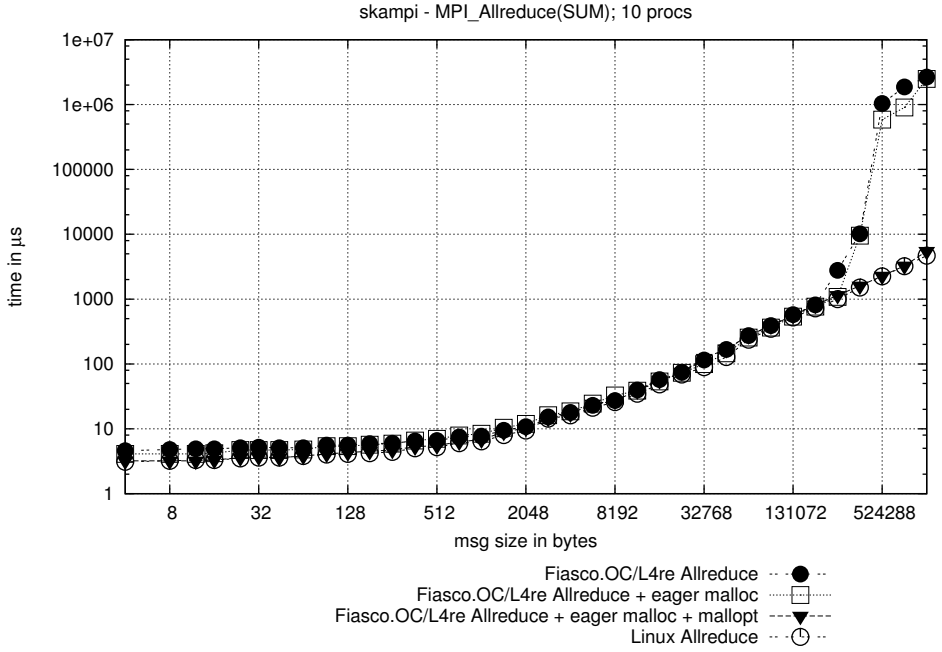

Figure 4.8: MPI_Allreduce() with 10 processes

broadcast the result of the reduce operation to all other processes in the communicator.

`MPI_Bcast()` runs fastest on Linux. Since the time gap between Fiasco and Linux remains visually constant on a logarithmically scaled diagram, it is likely to be dependent on message size. `Nemesis` ships its own `memcpy()` implementation (`MPIUI_Memcpy()`) and I doubt that differences in `libc` can be the source of such runtime differences. I believe placement and its effect on data locality, that affects memory bandwidth, provides a probable explanation.

`MPI_Reduce()` seems to be affected from paging for small message sizes. Other then that, the performance for this benchmark is about the same for all variations.

Figure 4.10 While `MPI_Scatter()` distributes data from the root process to the other processes in the communicator, `MPI_Gather()` collects buffers into the root buffer. Contrary to a broadcast, the data is usually not uniform.

When invoking `MPI_Alltoall()` every process sends distinct data to every other process. `MPI_Scan()` performs a prefix reduction with all processes of a group.[8] I believe the peak, when invoking `MPI_Alltoall()` for the first time, is due to a huge mapping (`mallopt()`) when touching memory for the first time. The little valley for `MPI_Scan()` at about 256 bytes is likely caused by a switch in the algorithm of `MPI_Scan()` to

[8] A prefix sum calculates by $r_i = \sum_{k=0}^{i} s_k$. For example MPI_Scan(MPI_SUM) on four processes 0..3 and their send buffers $\{s_0 = 1, s_1 = 2, s_2 = 3, s_3 = 4\}$ would result in values $\{r_0 = 1, r_1 = 3, r_2 = 6, r_3 = 10\}$ in the receive buffers.

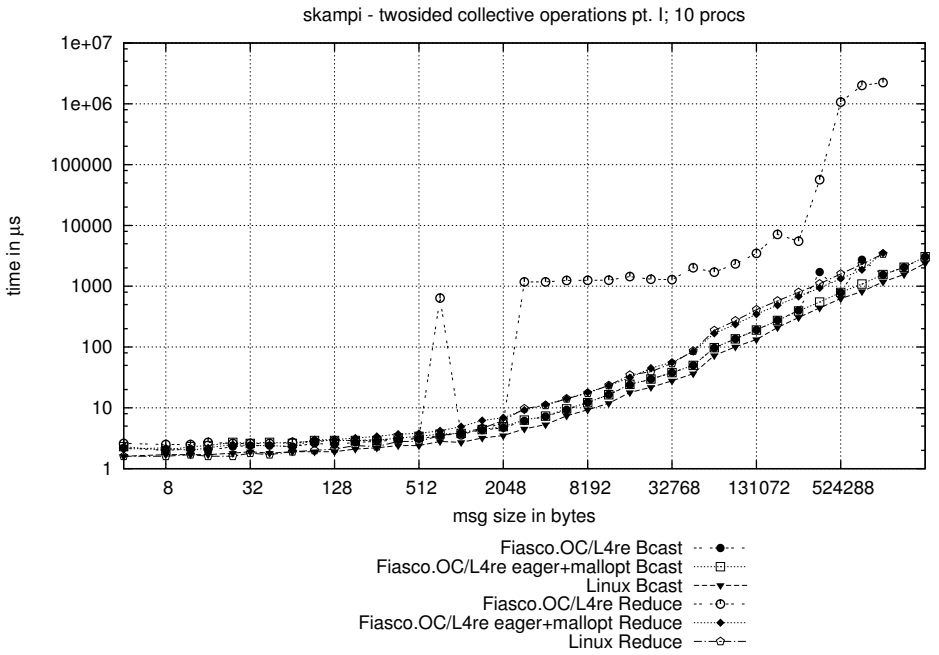

Figure 4.9: MPI_Bcast() and MPI_Reduce()

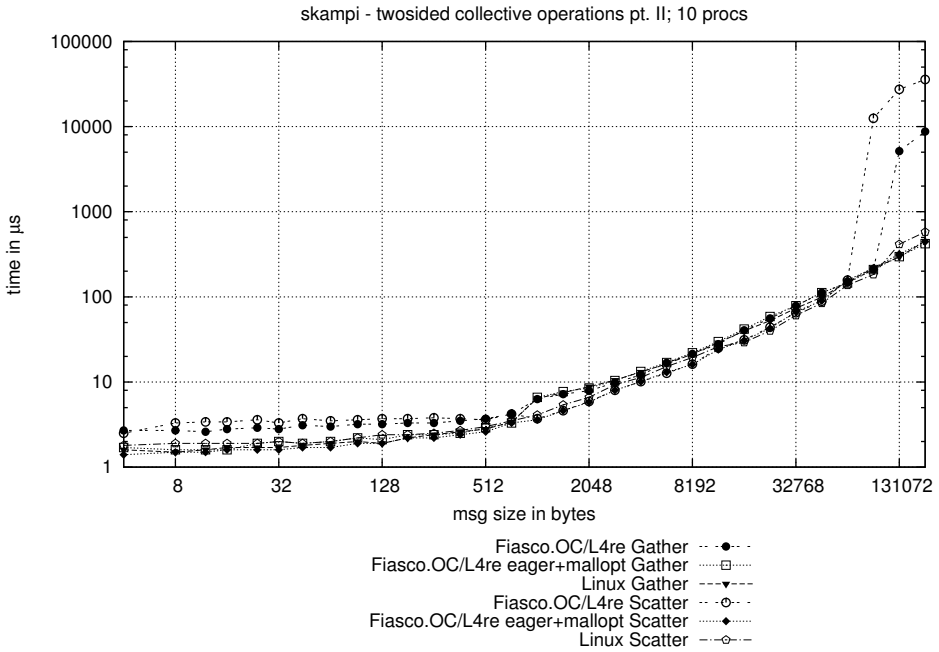

Figure 4.10: MPI_Gather() and MPI_Scatter()

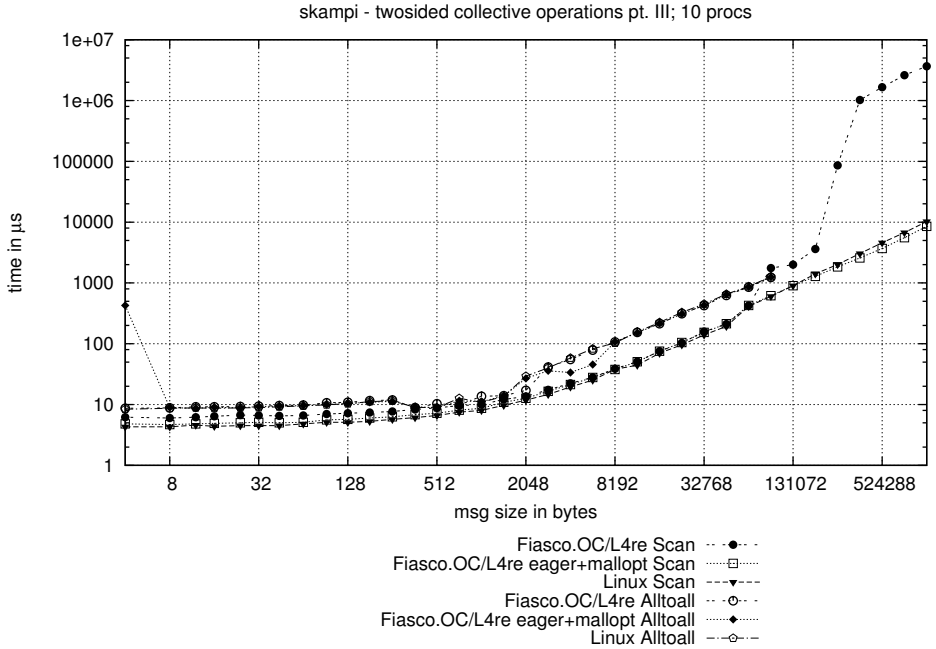

Figure 4.11: MPI_Scan() and MPI_Alltoall()

improve performance for larger message sizes. The implementation of `MPI_Scan()` uses not less then four different algorithms, depending on the message size and the size of the communicator. `MPI_Scan()` seems to run a little faster on Linux. This could a be related to placement and might hint at differing performance for library functions such as `malloc()`.

Figure 4.12 and 4.13 This are enlarged versions of the previous collective operations for **short message sizes**. I only include Fiasco.OC/L4Re eager_mapping + `mallopt()` variants to avoid large scaling due to paging.

4.3.2 One-sided Operations

Figure 4.14 `MPI_Put()` transfers data from the application buffer of the sender into a window (`MPI_Win_create()`) shared between all processes in the group of the communicator. `MPI_Put()` requires no matching call (such as a receive) in the destination. This benchmark was repeated multiple times for several (random) placements on both platforms. In analogy with the measurement methodology of `perftest` (Section 4.2.2.2), I chose the measurement with the best performance (minimum runtime) for Linux and Fiasco.OC/L4Re, to reduce placement effects. Each `mpptest` (ping-pong benchmarking tool contained in `perftest`) measurement was repeated 32 times on both platforms.

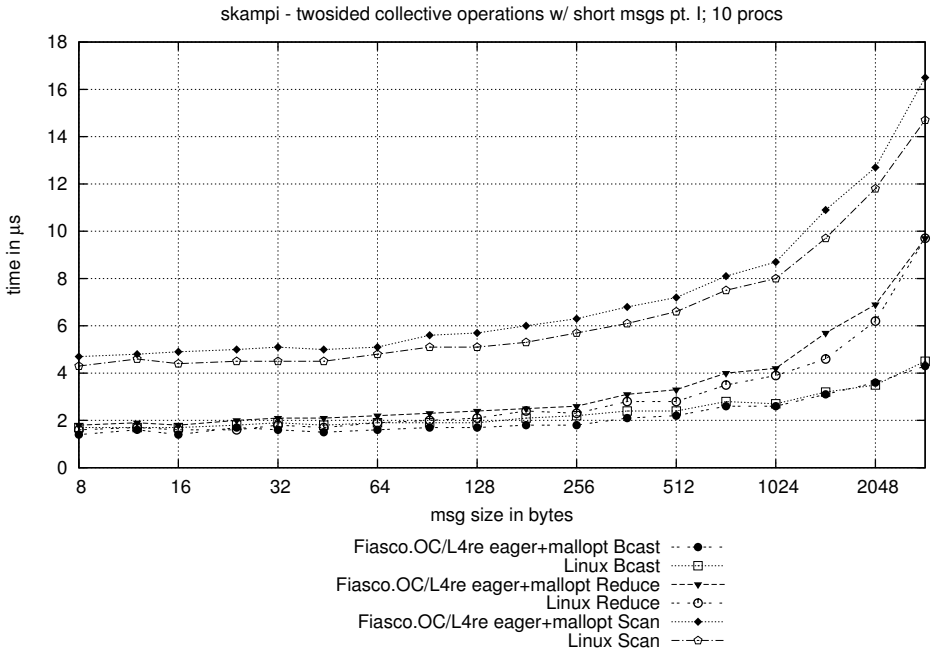

Figure 4.12: Detailed view for short messages: MPI_Bcast(), MPI_Reduce() and MPI_-Scan()

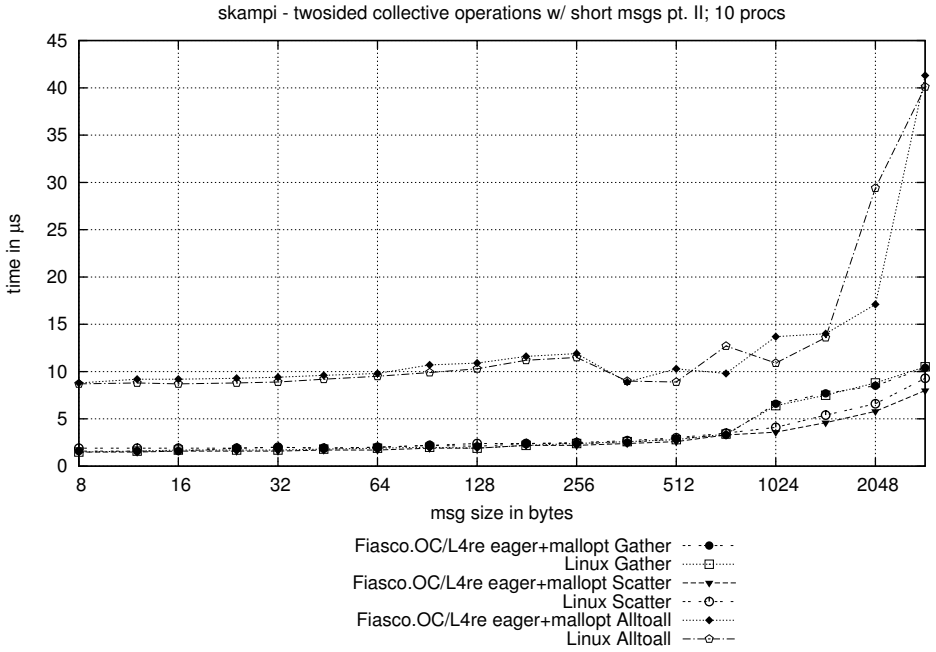

Figure 4.13: Detailed view for short messages: MPI_Gather(), MPI_Scatter() and MPI_-Alltoall()

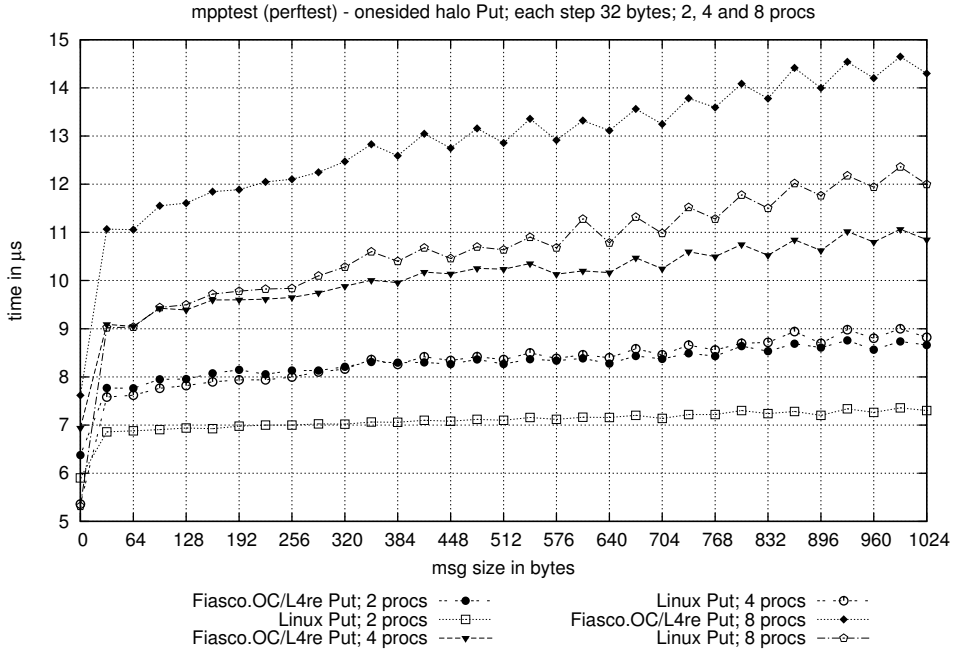

Figure 4.14: MPI_Put() with varying number of partners

Figure 4.15 `MPI_Get()` fetches data from target memory window into the origin buffer. Other than that, what applies to `MPI_Put()` also applies here.

`mpptest` offers a myriad of option combinations (e.g. window fence assertions and synchronization methods) to benchmark remote memory access (RMA) operations. I chose to only include two tests using basic active target synchronization. A call to `MPI_Win_fence()` begins the next (and ends the previous) communication epoch for a group of processes. Within this epoch, multiple RMA operations can be performed on all windows that belong to the same group as the window supplied to `MPI_Win_create()`. Each epoch ends with another call to `MPI_Win_fence()`. Both benchmarks were run using 10 processes, but with different numbers of partners taking part in the communication. That means, when 2 (4, 8) partners are used, only 2 (4, 8) partners perform actual RMA operations with their partner(s). The remaining 8 (6, 2) processes perform the window creation (not measured) and fence operations[9].

The two curves look like the sum of a linear function with a triangle wave. I think the triangle wave fraction is caused by the size of a cache-line (64 byte) and the increment to the message size (32 byte)[10].

[9] *mpptest* performs a *MPI_Allreduce(MAX)* on the measured elapsed time and not MIN after every measurement cycle. The minimum of all maxima is then selected from all measurements of a certain size.

[10] Given the memory address is not present in the cache and no other process attempts to either write or read from this memory address, then for every aligned write to 64 bytes in steps of 32 byte, the

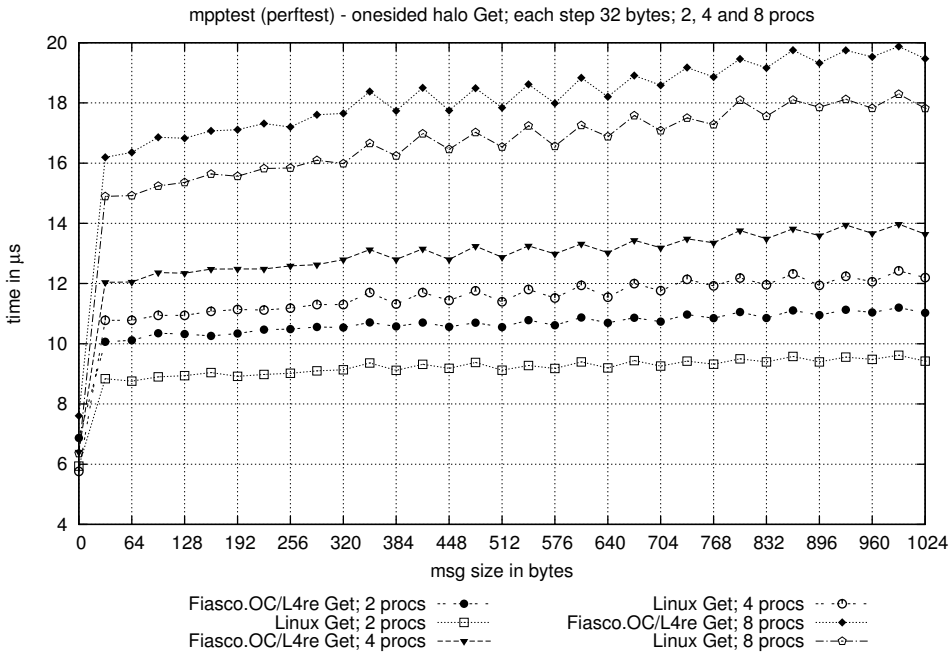

Figure 4.15: MPI_Get() with varying number of partners

Both RMA tests perform better on Linux. The offset seems to be independent of the message size for both platforms. The number of communicating partners has no effect on the offset for `MPI_Get()` but shows an increase for `MPI_Put()`. I have no good explanation for this behaviour, but believe that placement is not its cause. Inspecting the binaries exhibited no significant differences[11] in the way the MPICH library and `mpptest` were built. To me it appears, that the most probable source of this runtime differences is code from other libraries. Both operations rely on memory allocation taking place on the heap. It could be, that the `malloc()` implementation of `uclibc` (a `dlmalloc` variant) performs inferior compared to `malloc()` of GNU `libc` (`ptmalloc2`, based on dlmalloc with more fine-grained locking). When I was contemplating `malloc()` as a source of runtime differences (the case w/o page faults), functions used by `malloc()` also came to mind, especially locking functions of the `pthread` library.

On a last note, I want to discuss the question why a `malloc()` is invoked when calling RMA functions at all. The `Nemesis` communication subsystem uses `malloc()` to allocate and enqueue[12] RMA requests. When `MPI_Win_fence()` is called to end the communication epoch, all pending RMA operations are carried out.

first 32 bytes cause a write-miss (MOESI: Invalid -> Modified) and the second a write-hit (MOESI: Modified -> Modified). See [AMD12].

[11] Apart from function addresses and very few instructions (*nop*), the disassembled output appeared to be identical.

[12] *Nemesis* FIXME: RMA with short contiguous messages should bypass this queuing mechanism and requests should be allocated from a per thread memory pool.

Contrary to this, `MPI_Send()` is optimized for differing message sizes. To send short messages, a request packet is constructed on the stack and the message passed immediately (eager send/recv). Long messages are not transferred immediately (rendezvous send/recv). They require that a request packet is allocated from a memory pool, thereby bypassing `malloc()`.

I believe that this implementation difference between one- and two-sided operations explains why point-to-point and collective operations, that are composed of `MPI_send()`/`MPI_recv()`, have about the same performance on Linux and Fiasco.OC/L4Re whereas one-sided operations perform better on Linux.

5 Conclusion and Future Work

5.1 Conclusion

The MPICH library was successfully ported, tested and benchmarked on Fiasco.OC/L4Re. On the way to port the simple process manager `gforker`, several reusable artifacts, most notably `libendpoint`, an easy to use library that aids creating bidirectional pipes and `libspawner`, a library to spawn processes for the common case, were created.

MPICH on L4Re and MPICH on Linux were measured to have comparable performance on shared-memory, when the effect of slower memory management, namely paging, in L4Re is suppressed.

5.2 Future Work

5.2.1 Hydra and Slurm

The process manager that I ported is simple. It does neither have process binding, which is important to performance, nor many different ways to launch processes (e.g. `ssh`, `rsh`, `slurm`). It also lacks an integration of resource management[1].

I believe that porting `Hydra`, that possesses all of the above mentioned functionalities, could be an important step on the way to high performance computing clusters on top of Fiasco.OC/L4Re. To achieve this, `l4re_vfs` must provide an efficient and universal (e.g. different descriptor types) demultiplexing mechanism (`select()`/`poll()`).

5.2.2 Memory management

Memory management can be improved for the MPI use case. Linux `madvise(MADV_SEQUENTIAL)`-type page prefetching semantics or a more application specific paging algorithm can be implemented in the (or a specialized) region mapper in conjunction with the respective dataspace type. Implementing a region mapper with an additional interface that makes lazily unmapped regions feasible could improve the situation, especially when large regions of a similar proportion are continuously mapped and unmapped (e.g. temporary buffers of MPI operations). When the pager runs out of memory, it could evaluate information shared by each tasks region mapper, free

[1] Resource management is necessary on shared HPC systems where multiple MPI applications are run with a certain priority on a subset of the available nodes. This is achieved using a central instance, that manages where MPI processes are allocated. Examples of common resource managers are *slurm* and the *pbs* system.

pages from the selected (LRU) region and inform the respective region mapper, that an unused region has lost all its flex-page mappings and is therefore of no further use. More ways to tackle this problem are possible. I think most of them will turn out better then resorting to `mallopt()`. Solving this problem should have the highest priority.

5.2.3 Interconnect

High performance computing uses low latency/high throughput interconnects to communicate between different nodes. The protocol stack of such interconnects is either implemented on the hardware itself or by low latency user-space protocol stacks. In either way kernel entries and exits as well as context switches are avoided to achieve latencies of $1 - 3\mu s$. Variants of InfiniBand and Gigabit Ethernet seem to be the most promising candidates for this.

5.2.4 Fortran and Libraries

Since Fortran has arrived in the Fiasco userland, Fortran bindings can be added to MPICH. This opens the possibility to port highly optimized algorithmic libraries. Most noteworthy are BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra Package).

5.2.5 Userland and POSIX Compatibility

The L4Re userland lacks support for many aspects of the POSIX API. This makes it harder to port applications that use POSIX functions to L4Re. Improving support for signal handling, process creation and handling, pipes, file operations (including sockets and demultiplexing back-ends) could dramatically improve the speed with that applications can be ported to L4Re and thereby result in a richer userland.

Index

Bibliography

[AMD12] AMD. Amd64 architecture programmer's manual volume 2: System programming, 24593—rev. 3.22, 2012. 40

[BBG+10] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Krishna, E. Lusk, and R. Thakur. Pmi: A scalable parallel process-management interface for extreme-scale systems. *Recent Advances in the Message Passing Interface*, pages 31–41, 2010. 16

[BKM+10] Tom Budnik, Brant Knudson, Mark Megerian, Sam Miller, Mike Mundy, and Will Stockdell. Blue gene/q resource management architecture. In *Many-Task Computing on Grids and Supercomputers (MTAGS), 2010 IEEE Workshop on*, pages 1–5. IEEE, 2010. 10

[BM06] Darius Buntinas and Guillaume Mercier. Implementation and shared-memory evaluation of mpich2 over the nemesis communication subsystem. In *Proceedings of the Euro PVM/MPI Conference*. Springer, 2006. 17

[Fly72] M. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput.*, C-21:948+, 1972. 3

[For97] High Performance Fortran Forum. High performance fortran language specification, 1997. 3

[GD12] W. Gropp and MPICH Developers. Pmi v2 api, 2012. [Online; accessed 24-August-2012]. 7

[GL99] William Gropp and Ewing Lusk. Reproducible measurements of mpi performance characteristics. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 681–681, 1999. 28

[GLS99] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI, 2nd Edition: Portable Parallel Programming with the Message Passing Interface*. Scientific and Engineering Computation. MIT Press, 1999. 3

[GT07] William D. Gropp and Rajeev Thakur. Revealing the performance of mpi rma implementations. In *PVM/MPI'07*, pages 272–280, 2007. 5

[HL10] Gernot Heiser and Ben Leslie. The okl4 microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, pages 19–24. ACM, 2010. 9

[HSJ+06] Wei Huang, Gopalakrishnan Santhanaraman, H-W Jin, Qi Gao, and Dhabaleswar K Panda. Design of high performance mvapich2: Mpi2 over infiniband. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 43–48. IEEE, 2006. 17

[Int13] Intel. Ia-64 and ia-32 architectures software developer's manual. *Volume-1: Basic Architecture*, 2013. 3

[Jur06] Matthias Jurenz. Vampirtrace software and documentation. *ZIH, Technische Universität Dresden, http://www.tu-dresden.de/zih/vampirtrace*, 2006. 4

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009. 9

[Lie92] Jochen Liedtke. Clans & chiefs. In *In 12. GI/ITC Fachtagung Architektur von Rechnersystemen*, pages 294–305, Kiel, 1992. 8

[Lie93] Jochen Liedtke. Improving ipc by kernel design. *SIGOPS Oper. Syst. Rev.*, 27(5):175–188, December 1993. 8

[Lie96a] Jochen Liedtke. L4 reference manual - 486, pentium, pentium pro, 1996. 8

[Lie96b] Jochen Liedtke. Toward real microkernels. *Commun. ACM*, 39(9):70–77, September 1996. 7

[LW09] Adam Lackorzynski and Alexander Warg. Taming subsystems: capabilities as universal resource access control in l4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, IIES '09, pages 25–30, New York, NY, USA, 2009. ACM. 9

[Mes09] Message Passing Interface Forum. Mpi: A message-passing interface standard, version 2.2. Specification, September 2009. 4, 5, 6, 32

[MG02] Bernd Mohr and Michael Gerndt. *Parallel Programming Models, Tools and Performance Analysis*, volume 10 of *NIC Series*. John von Neumann Institute for Computing, Jülich, 2002. 4

[Ope11] OpenMP Architecture Review Board. Openmp application program interface. Specification, 2011. 3

[SAB+08] Edi Shmueli, George Almasi, Jose Brunheroto, Jose Castanos, Gabor Dozsa, Sameer Kumar, and Derek Lieber. Evaluating the effect of replacing cnk with linux on the compute-nodes of blue gene/l. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 165–174. ACM, 2008. 11

[SGK+85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. In *Proceedings of the Summer USENIX conference*, pages 119–130, 1985. 15

[Spi11] M. Spiegel. *Cache-Conscious Concurrent Data Structures*. PhD thesis, University of Virginia, 2011. 4

[SS75] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975. 19

[STJ+08] Subhash Saini, Dale Talcott, Dennis Jespersen, Jahed Djomehri, Haoqiang Jin, and Rupak Biswas. Scientific application-based performance comparison of sgi altix 4700, ibm power5+, and sgi ice 8200 supercomputers. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 7:1–7:12, Piscataway, NJ, USA, 2008. IEEE Press. 4

[top] TOP500 supercomputer site. 10

[Wal92] D. Walker. Workshop on standards for message-passing in a distributed memory environment. Technical Report TM-12147, Oak Ridge National Laboratory, August 1992. 6

[YTR+87] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, and Jeffrey Eppinger. *The duality of memory and communication in the implementation of a multiprocessor operating system*, volume 21. ACM, 1987. 8