UPPSALA
UNIVERSITET

UPTEC IT 14 020
Examensarbete 30 hp
November 2014

Analysis and Design of High Performance Inter-core Process Communication for Linux

Andreas Hammar

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Analysis and Design of High Performance Inter-core Process Communication for Linux

Andreas Hammar

Today multicore systems are quickly becoming the most commonly used hardware architecture within embedded systems. This is due to that multicore architectures offer more performance at lower power consumption costs than singlecore architectures. However, since embedded systems are more limited in terms of hardware resources than desktop computers, the switch to multicore has introduced many challenges. One of the greatest challenges is that most existing programming models for interprocess communication are either too heavy for embedded systems or do not perform well on multicore. This thesis covers a study on interprocess communication for embedded systems and proposes a design that aims to bring the two most common standards for interprocess communication to embedded multicore. Furthermore, an implementation that achieves efficient inter-core process communication using zero-copy shared memory is implemented and evaluated. It is shown that shared memory utilization yields much higher performance than conventional communication mechanisms like sockets.

Handledare: Detlef Scholle
Ämnesgranskare: Philipp Rümmer
Examinator: Roland Bol
ISSN: 1401-5749, UPTEC IT 14 020

Contents

Acknowledgments

My deepest gratitude goes to my supervisors Detlef Scholle and Cheuk Wing Leung for their guidance and support throughout the work this master thesis. Also I want to thank Alten and all its employees for providing me with a good working environment for my thesis. A special thanks goes to Anton Hou and Fredrik Frantzén for being part of a great team to work in. I would also like to thank my reviewer Philippe Rümmer for ensuring high quality of my report.

Abbreviations

Abbreviation	Definition
API	Application Programming Interface
CPU	Central Processing Unit
GPC	General Purpose Computer
HPC	High Performance Computing
IPC	Interprocess Communicaiton
MCAPI	Multicore Communications Application Programming Interface
MPI	Message Passing Interface
NUMA	Non Uniform Memory Access
OTF	Open Trace Format
OS	Operating System

Chapter 1

Introduction

1.1 Background

Embedded systems continually grow more complicated, which creates an increasing need for more computing power in such systems. To support this need, the industry is shifting from centralized single-core systems towards many-core solutions that can provide more computing power, in a way both more reliable and power efficient. However, due to Amdahl's law there is a limit to how much performance can be gained from running parts of an application in parallel [1]. This is partually due to that parallel tasks must be able to synchronize. For synchronization, communication between the tasks is needed and the speed of this communication greatly affects the performance gained from parallelization.

To provide efficient embedded systems, both hardware and software must be able to adapt to various situations such as limited battery life or hardware resources. This has led to that embedded systems hardware is getting increasingly more complex. To simplify the development of efficient applications on many-core systems, the complexity of the hardware structure must be hidden from the application developer. The abstraction layer must not only provide a simple interface for application developers, but also be dynamically reconfigurable to enable applications to run efficiently on a variety different hardware configurations. A solution that can provide this flexibility is the use of middleware which acts as an abstraction layer between the running application and the underlying hardware and operating system configuration.

A very important feature for efficient hardware utilization of many-core systems is to allow processes to share data between each other. Currently, the most efficient way to allow on chip interprocess communication (IPC) is through the use of shared memory. However applications that use memory sharing between processes are complicated to develop since such programming models must preserve data coherence and synchronization between parallel tasks. Another common way to share data between processes which is much simpler is to use message passing, which is a technique that involves a lot of redundant copying of data between processes. Therefore message passing is less memory efficient and has lower performance than the use of memory sharing. The ideal solution would be to develop an IPC protocol with the simplicity of message passing that utilizes shared memory. LINX [2] developed by Enea AB is a good example of a powerful, open source and platform independent tool for message passing that supports zero copy shared memory between local processes, where local refers to the processes being run on the same CPU node. LINX also supports a multiple number of heterogeneous types of links for remote IPC. However currently there is no utilization of shared memory support for the Linux implementation of LINX. This can be a performance bottleneck for multicore processors as applications tend to have a need for frequent on-chip IPC.

This master thesis project has been conducted at Alten AB in collaboration with Uppsala University. The thesis project will be part of the CRAFTERS [3] project which aims to create an innovative ecosystem from application to silicon and funded by ARTEMIS [4].

1.2 Problem Statement

The main goal of this master thesis is to design and implement an efficient solution for on-chip interprocess communication using message passing on Linux. The message passing solution will be able to efficiently utilize the shared memory between cores on multicore processors. This thesis will base its implementation design on the results from a previous master thesis conducted at Alten where a design for using zero-copy shared memory communication was suggested.

The goal of this thesis is to cover the following topics:

1. **Survey of existing on-chip IPC solutions**
   The goal of covering this topic is to get a clear picture of what solutions for on-chip IPC that already exists and identify what disadvantages that these solutions have on multicore. The knowledge gained from this survey will lay the foundation for proposing a design that aims to solve some of these challenges.
2. **Design of efficient on-chip IPC for embedded systems on Linux**
   The designed solution must emphasize the following key features:
   - Shared memory between cores must be efficiently utilized.
   - The communication abstraction must be simple to use and provide a high level abstraction of the underlying implementation.
   - Low communication overhead suitable for embedded systems.
3. **Implementation of efficient IPC**
   The goal of the implementation part of this thesis is to provide a reference implementation of the design. The implementation may not be able to include all features of the design, but is able to demonstrate the performance of the design.

1.3 Relation to Previous Work

The implementation developed in this thesis will study the suggested design from a previously conducted thesis at Alten AB [5] which analyzed and proposed a design for zero-copy message passing on Linux. To be able to create an efficient implementation from that design, several other factors must also be taken into account, such as traceability, existing standards, useability, robustness and performance. Several other message passing implementations targeting embedded systems that utilize the benefits of shared memory exist [6–9]. The results from those implementations will be taken into account for the design of this implementation.

1.4 Method

The project goals have been set and evaluated according to the requirements from UKÄ for a master thesis [1]. The project will consist of an academic study to provide background for a good design, and then an implementation phase, each of which will span about half of the project. The academic study will cover a background study and analysis on message passing solutions. Also the memory management in the Linux operating system will be studied in order to determine which solution is suitable for implementation. The practical part will consist of developing an implementation of zero-copy shared memory message passing on Linux in C. The implementation will then be compared to existing IPC mechanisms to evaluate the performance.

[1] http://www.uk-ambetet.se/utbildningskvalitet/kvalitetenpahogskoleutbildningar.4.782a298813a88dd0dad800010221.html

1.5 Use Case

1.5.1 Team Goal

The practical part will also consist of developing components of a middleware developed in a team of three students that combines the knowledge gained from all three theses. To demonstrate the features of the middleware, an application is developed that performs on-line face recognition from a video stream. The contributions from this thesis will be able to provide the middleware with an efficient IPC mechanism that is able to transfer the data from the video stream at a high rate between the system components. The team will be working using scrum and test-driven development where the system tests will be modeled using a modeling tool that utilizes UPPAAL [10].

1.5.2 Team Workflow

The team worked in an agile software development approach using scrum and test driven development. The development cycles were kept very short, where every scrum sprint was kept to one week. Each week the team met up with the project supervisors and presented their progress to ensure high quality in the work and that the project progresses as planned.

1.5.3 Development Tools

Several tools were used to help the development of the applications for all three of the team's master theses. The team source code was maintained using the version control tool git [11]. Furthermore to automate the running of software tests and static analysis tools a continuous integration running JenkinsCI [12] was used. The server would monitor the version control repository of the source code and periodically run a series of tools, both static analysis tools like Cppcheck and unit tests developed by the team itself.

1.6 Delimitations

The thesis will only study on-chip message passing, and for the implementation utilize existing functionality in open source alternatives for communication between processor nodes. In particular techniques that target embedded systems using shared memory will be studied in depth. The project work is limited to a time period of 20 weeks.

1.7 Contributions

The contributions of this thesis is a design that is simple to use, compatible with the most relevant existing standards for message passing, while still achieving high performance and a low communication overhead.

1.8 Report Structure

The remaining content of the report is divided into three main sections. Chapters 2-5 will cover the background study of and analysis of interprocess communication. Chapter 6 will describe the design of the solution proposed in this thesis. Chapters 7 and 8 will describe what is covered in the implementation, while chapters 9 and 10 will cover the evaluation of and conclusions from the implementation.

Chapter 2

Background: Interprocess Communication on Multicore Systems

2.1 Introduction

Moore's law continues, but processor frequencies have stopped increasing. This is mainly caused by the fact that processors have started hitting the power wall, where the amount of power needed and heat generated by increased processor frequencies outweigh the performance gains. Instead the computing industry has started shifting towards multicore solutions which provide more computing power using less power than single core architectures. However there is a limit to how much performance can be gained from the parallelization of an application, dividing its computation across multiple cores. This limit is known as Amdahl's law [1]. To cope with Amdahl's law and be able to utilize the system hardware efficiently on multicore architectures, new programming models must be developed that support parallelism. This situation has been present for the past couple of years within the desktop and high performance computing(HPC) environment. Now however the problem is also starting to appear within the embedded world due to that there is a lot of power and efficiency that can be gained when switching from current decentralized systems to multicore.

Although the current situation is similar to what has happened in the HPC environment over the past couple of years, the requirements on new programming models that support parallelism are different for embedded systems. This is mainly due to the fact that embedded multicore systems differ from the multiprocessors seen in HPC systems by their tendency to contain multiple heterogeneous processing elements, non-uniform memory structures and non-standard communication facilities such as Network-on-Chip communication [13].

2.2 The Importance of Efficient Interprocess Communication

Interprocess communication(IPC) is the technique of sharing information between parallel processes. This technique is an important part of parallel programming since in most applications parallel tasks must communicate between each other frequently for their computations to be meaningful. For applications that use communication between tasks frequently it is crucial that the used interprocess communication mechanism is as fast as possible. If the interprocess communication would be slow, the results could be that parallel applications run slower than their sequential counterpart due to the performance bottleneck of the communication.

One of the greatest challenges with interprocess communication is to synchronize the data shared between parallel tasks. Currently the most common way to achieve synchronization is to use locks. Locks however may introduce a lot of waiting in program execution and along with the increasing number of cores it becomes more and more difficult to avoid deadlocks.

2.3 Message Passing or Shared Memory

The two currently most commonly used programming models for interprocess communication are message passing and shared memory. Both models present an efficient solution to both task communication and synchronization. Although the abstraction presented by message passing is very different from that of shared memory, their underlying implementations can still utilize system resources in the very same way. For example a message passing API presents the user with a function call where the user transmits a message from one endpoint to another within the system, while in fact the message passing implementation will simply write the message to a shared memory segment between the two endpoints. As such the two models can often be seen as a duality of each other, which has caused a long ongoing debate about the differences between the two models [14]. However this does not mean that one of the models can simply be chosen instead of the other, instead the strengths and weaknesses of both models should always be considered in the design of a system utilizing one of the two. The following sections will describe these two programming models in further detail.

2.4 Shared Memory

Shared memory is a technique where processes are allowed to share the access to the same memory segment between each other. It is currently the fastest available IPC technique due to that once a shared memory segment has been allocated, the kernel does not need to be involved when passing data between processes [15]. However since the kernel is not involved when passing data, the communicating processes must themselves manage data synchronization. This often puts the responsibility of synchronization on the application developer, making it more complicated to develop robust applications.

What makes shared memory IPC more efficient than other IPC mechanisms is that data does not need to be copied from one process memory space to another. Instead a memory space that is shared between the communicating processes is created. This way IPC that uses shared memory avoids many costly memory operations.

2.4.1 Data Coherence

One of the greatest challenges for shared memory programming models on multicore systems is maintaining cache coherence between communicating processes running on different cores. This is caused by the common use of Non-Uniform Memory Access(NUMA) memory architectures in multicore systems. The problem that shared memory programming models face on NUMA architectures is that shared data must be synchronized between the local caches of the system cores. The problem that occurs is when one of two processors that share data updates that data within their local cache, the other processor will still have the old copy of that data in their cache. To solve this problem the system must ensure that both processors always have the same copy of shared data within their cache. Maintaining consistency between local caches introduces memory overhead that may greatly reduce the performance of the system. As core counts keep increasing, the cost of maintaining coherence also increases. In fact, some researchers have even suggested that cache coherence will no longer be feasible across a single multi-core chip, or that individual cores may perform better in the absence of coherence [16].

2.4.2 Synchronization

As mentioned earlier, processes communicating using shared memory must themselves ensure data synchronization and the most common way to do this is to use locks or semaphores. Locks are binary flags that prevent two or more processes from reading or writing to the same shared memory at the same time. Semaphores are similar to locks with the difference that instead of being a binary lock giving access to one process at a time, there can be an arbitrary amount of processes that may acquire access to the shared resource at the same time.

Synchronization using locks or semaphores may introduce waiting in programs resulting in that processes spend most of their time performing redundant waiting. The use of locks also introduces the possibility of deadlocks within an application, which with increasing parallelism becomes very difficult to avoid. A deadlock occurs when two or more processes are waiting for each other to finish their use of a shared resource while each one has locked a resource the other process needs to continue execution. This causes both processes to wait infinitely for each other, stalling the system.

2.4.3 Distributed Shared Memory

Distributed shared memory is a solution that aims to provide the shared memory abstraction in a system of distributed nodes that do not share the same local memory. What this abstraction provides is a memory segment that is replicated and synchronized on each of the communicating distributed nodes. These solutions typically uses message passing to synchronize the shared memory segments and thus it tries to combine the ease of use of shared memory, and the distributed performance of message passing.

2.5 Message Passing

Message passing is an IPC programming model where communicating processes share data through sending messages. The main goal of a message passing interface is to provide a transparent communication interface where all the name resolution and shared data between processes is managed by the underlying message passing implementation. Hence message passing provides a high level abstraction of the underlying hardware.

The message passing can either be synchronous or asynchronous. In synchronous message passing the communicating processes synchronize at some point in their execution and exchange data. This type of message passing is often robust and ensures data synchronization between tasks, however it also introduces busy waiting where tasks have to wait for the task it wants to communicate with to reach its synchronization point.

In asynchronous message passing the sending process does not need to wait for the receiving process to reach its synchronization point. Instead the sending process passes the message directly to the underlying Message Passing Interface(MPI) which makes sure that the message gets received at the other end. In many existing applications this is done by having a shared message buffer between the two communicating processes. This is very similar to how letters are sent through the postal service where the sender puts the letter into a mailbox and can then go on with their life while the postal service takes care of delivering the letter. Similarly the sender of an asynchronous message can continue its execution immediately after the message has been sent, regardless of whether the receiver has received the message or not.

2.6 Message Passing Vs Shared Memory

This section addresses some of the important benefits and weaknesses between using one programming model over the other. It is important to note though that there is no definitive answer as to which programming model that is better than the other.

While there are benefits of both models, shared memory programming models are both preferred from a usability standpoint and has a much lower learning curve [17]. However as hardware complexity is increasing in multicore architectures, it is becoming more difficult to provide a simple shared memory abstraction. Shared memory models also suffer more from problems with cache coherence in NUMA architectures since message passing communication is very explicit and there are mostly only one writer to each memory buffer. As mentioned earlier in subsection 2.4.1 future systems might not even be using cache coherent memory which raises an interesting question on the performance of shared memory in such systems.

In [18], it was pointed out that the trend towards using high level languages that is compiled to specific parallel targets suggests that the difference in ease of use between message passing and shared memory will be eliminated. Thus it is often simply a matter of choice and personal preference that are involved when deciding between which of the two programming models to use for a specific application.

The benefits of both programming models has given rise to several implementations that aims to combine the benefits of both programming models. These implementations can in general be divided into two categories based on which programming model that the interface they provide resembles. One category being message passing libraries that utilize shared memory techniques which will be covered in the following chapter. The other category is libraries for distributed shared memory that use message passing to maintain memory coherence across distributed nodes.

Chapter 3

A Survey of Message Passing Libraries

3.1 Introduction

This chapter will cover several existing message passing libraries that aims to support a wide variety of different hardware configurations and communication media types. The designs studied in this chapter will be analyzed to how well they fit the requirements for an IPC library on embedded multicore. The design choices of these implementations will lay the foundation for the design of a new message passing API targeted specifically for embedded multicore systems.

3.2 Requirements on a Message Passing API For Embedded Systems

As mentioned in section 2.1 embedded multicore systems have a tendency to have a more complex and heterogeneous architecture than general purpose computers (GPC). This makes the requirements for software libraries targeting embedded multicore systems different from those targeting GPC:s.

1. **Small Memory Footprint**
   The amount of available memory on embedded systems is often very limited. Therefore the library of any API aimed at embedded systems must have a small memory footprint. Furthermore an efficient IPC library on a multicore processor must be able to fit inside the local cache of each processor core. This puts a huge constraint on the memory footprint of a message passing library targeting embedded multicore systems.

2. **Modular Design**
   Within embedded systems, hardware changes almost as quickly as software. This means that the amount of different hardware platforms that a new message passing API must support is massive. Therefore it is crucial that the design of a message passing library targeting embedded multicore systems is modular so that it is easy to port to a new hardware platform.

3. **Transparent Abstraction of Underlying Hardware**
   For a message passing library to become widely adopted it must provide a good abstraction of the underlying hardware. As hardware is becoming significantly more complex this requirement becomes more important. Furthermore, the rapid changes in hardware suggests that the abstraction must be high level so that the library itself configures how to utilize the system resources efficiently. If that task would be left for the developer, the hardware specific optimizations made by the developer on one platform might need to be rewritten when porting the application to a

new hardware platform.

4. **Compatibility with Existing Legacy Code**
   As mentioned in [13] the integration of existing legacy code within embedded systems is very important due to the high cost of redeveloping and recertifying systems. This implies the importance of compatibility with existing legacy code for new programming models developed targeting embedded systems.

3.3 Message Passing Interface (MPI)

MPI is a specification standard for message passing libraries. The specification is based on the consensus of the MPI Forum, which has over 40 participating organizations [19]. The main goal of MPI is to provide a highly transparent message passing interface that can provide high performance communication on any type of heterogeneous or homogeneous hardware configuration. The current MPI version 3.0 is a major update to the standard including both blocking and nonblocking operations [20]. Currently there are many available implementations of MPI, both freely available, open source versions and commercial products.

MPI is mainly aimed at high performance computing systems and as shown in [8], MPI is not suitable for multicore processors due to its high memory overhead. Instead a more lightweight solution should be used on multicore systems.

3.3.1 Open MPI

Open MPI is a widely used and open source implementation of the MPI standard. It provides a high performance message passing interface that supports many different platforms. Open MPI is based on and combines the experience gained from the LAM/MPI, LA-MPI and FT-MPI projects [21]. From a communications perspective, Open MPI has two main functional units, the Open Run-Time Environment (OpenRTE) and the Open MPI communications library [22]. The system resources are divided into cells of communication units that share common characteristics. For example a set of 4 homogeneous cores on the same chip could form a cell in a distributed computing cluster of many processor chips.

OpenRTE

OpenRTE is a complete runtime environment for distributed HPC applications. It provides a transparent interface with automatic configuration and management of hardware resources. This allows a user to take advantage of the system hardware effectively without writing any system dependent code. Currently OpenRTE provides support for a multitude of different heterogeneous and homogeneous hardware configurations. The design of OpenRTE was originally part of the Open MPI project but has later taken of into its own effort [22]. Some of the key design concepts of OpenRTE is [23]:

- **Ease of Use** An application should be able to execute efficiently on a variety of different hardware and should be able to scale to increasingly larger sizes without code modification.
- **Resilience** The run-time should never fail, even when encountering application errors. Instead it should gracefully terminate the application and provide an informational error message.
- **Scalability** The run-time environment must be able to support applications consisting of several thousands of processes that operate across a large number of processing units.
- **Extensibility** The run-time environment must support the addition of new features and the replacement of key subsystems new ones that use a different approach.

Open MPI Communications Library

The Open MPI communications library aims to give a full implementation of the MPI 3.0 message passing standard. It provides support for a wide range of different interprocess communication techniques and transportation media. Open MPI has a very modular design where each supported transportation type is handled by its own module which each provide a framework for communicating across a specific transportation media. When an application process wants to establish a connection with another process, all frameworks are queried to see if they want to run the process and the module with highest priority that wants to manage the connection will be selected [21].

3.3.2 MPICH

MPICH is a widely used and highly optimized open source implementation of MPI. It is as of November 2013 along with its derivatives used in 9 out of the top 10 super computers in the world [24]. It uses a layered design and aims to achieve portability to a large variety of platforms. The topmost layer, called the ADI3 is what presents the MPI interface to the application layer above it [25]. The second layer, called the device layer implements the point to point communication. This layer also implements the platform specific part of the MPICH communication system. Therefore MPICH can be ported to a new communication subsystem by implementing a device [25]. The standard implementation of MPICH comes with the CH3 device which is a sample implementation of a MPICH device. The CH3 device gives the possibility of using many different communication channels that each provide communication across a specific media type. Currently the default channel in CH3 is Nemesis which supports many different communication methods [26], however users may choose to implement their own communication channel and interface with the CH3 device.

3.4 MCAPI

MCAPI is a message passing library specification by the Multicore Association that aims to deliver an efficient and scalable interprocess communication library for embedded multicore devices. The MCAPI specification is mainly targeted at inter-core communication on a multicore chip and is based on both the MPI and socket programming APIs, which where targeted primarily at inter-computer communication [27]. Since MCAPI focuses on providing a low footprint and latency for multicore communication only it is not as flexible as MPI or sockets. However due to its low footprint it is much more suitable for embedded devices than MPI and as shown in [7] it achieves much higher performance than MPI on multicore processors. Also MCAPI provides support for communication on cores that runs without any operating system making it highly portable [27].

MCAPI provides inter-core communication using three different communication types including connectionless messages, packet channels and scalar channels which are visualized in figure 3.4. The three communication types are strictly prioritized. For example a process that has established a connection with another process using a packet channel may not communicate using messages between each other.

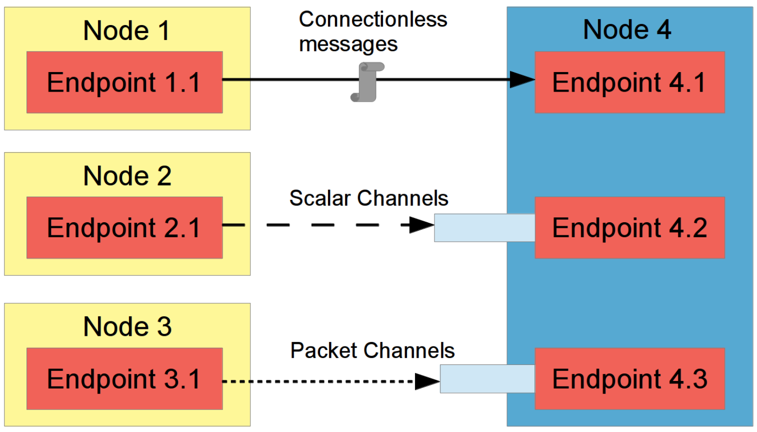


Figure 3.1: MCAPI message types

3.4.1 Messages

Message communication is the most flexible out of the three protocols provided by MCAPI. Messages provide a connectionless communication interface for applications that can be either blocking or non-blocking. The only requirement is that the application must allocate both the send and receive buffers.

3.4.2 Packet Channels

Processes may communicate through packet channels by first establishing a connection to each other. The benefit of first establishing a connection is that the message header may be reduced significantly [27]. Similarly to message passing, packet channels communicate using buffers. The receiving buffers are allocated by the MCAPI system and provides a unidirectional communication buffer between two processes. The application must however allocate the sending buffer by itself.

3.4.3 Scalar Channels

Scalar channels are very similar to packet channels but are customized to sending a stream of data efficiently between two communicating processes in a producer consumer fashion.

3.4.4 OpenMCAPI: An Open Source MCAPI

Open MCAPI [28] provided by Mentor Graphics is an open source full implementation of the MCAPI standard. It is designed for portability and is scalable down to low-resource embedded systems.

3.5 MSG

MSG presented in [9] provides an efficient message passing implementation for embedded systems that focus on portability and performance. The library implements a subset of required functions from the MPI standard and therefore provides some compatibility with already existing MPI applications. The subset of MPI functions was selected on the basis of the 14 most frequently used functions from 14 programs out of the Lawrence Livermore National Laboratory benchmark suite that was derived from [29]. The selected functions were divided into four categories, which can be seen in figure 3.2.

Furthermore the figure shows the number of calls and the occurrence of the functions calls in the 14 benchmarks. The category one-sided operations was added as an optional addition for performance considerations and provide function calls for direct memory access.

MPI Call	Description	Average Calls in 14 LLNL Benchmarks	Occurrence in 14 LLNL Benchmarks
Resource management functions			
MPI_Init	MPI initialization	N/A	N/A
MPI_Finalize	MPI finalization/cleanup		
Point-to-point operations			
MPI_Send	blocking send	80,337	11
MPI_Recv	blocking receive	53,648	
MPI_Isend	Non-blocking send	222,527	
MPI_Irecv	Non-blocing receive	246,530	
MPI_Test	Communication request	N/A	
MPI_Wait	Wait for request	65,881	
Collective Operations			
MPI_Barrier	Barrier synchronization across all members of a group	56	10
MPI_Bcast	Broadcast message from one member to all members of a group	2,067	
MPI_Scatterv	Scatter data from one member to all members of a group	N/A	
MPI_Gatherv	Gather data from all members to one member of a group	284	
One-sided Operations			
MPI_Put	Remote memory write	N/A	N/A
MPI_Get	Remote memory read		

Figure 3.2: Subset operations of MPI

Looking at figure 3.2 it seems that blocking message calls are more common than non-blocking calls. This means that although many lightweight implementations, such as LINX described in the next section, do not explicitly implement blocking calls it is required by a large number of existing MPI implementations.

3.6 LINX

LINX developed by Enea is an open source library for high performance interprocess communication. It provides a simple message passing protocol that abstracts both node locations and communication media from the developer. It currently supports both Enea OSE, OSEck and Linux operating systems and more are planned to be supported in the future. LINX provides a transparent message passing protocol that supports many different heterogeneous system configurations.

3.6.1 Key Features

LINX is designed for high performance and due to its small footprint is able to scale well down to microcontrollers [30]. It delivers reliable interprocess communication for a large variety of different hardware configurations. Furthermore LINX supports a large variety of different transportation media which makes it easy to configure even for systems using a mix of various connection media types.

3.6.2 System Overview

The Enea LINX protocol stack has two layers, the RLNH and the Connection Manager (CM) layers. The RLNH layer layer corresponds to the session layer in the OSI model and implements IPC functions such as name resolution of endpoints [30]. The connection manager corresponds to the transport layer in the OSI model and implements reliable in order transmission over any communication medium. There is also a link layer corresponding to the layer in the OSI model with the same name and implements OS and transportation media specific end to end delivery over the communication media. An overview of all these layers is shown in figure figure 3.3.

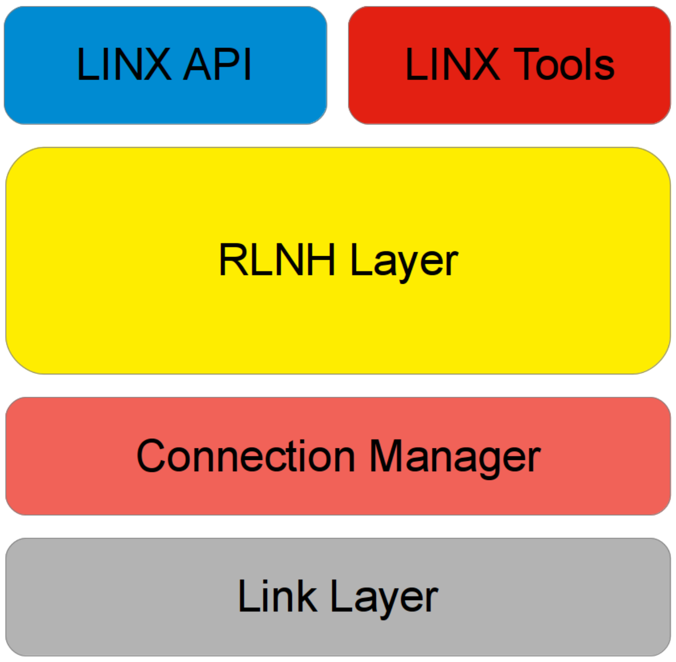


Figure 3.3: LINX Architecture [30]

The RLNH Layer

The RLNH is responsible for resolving remote endpoint names and for setting up and removing local representations of remote endpoints [30]. It is responsible for setting up and tearing down connection managers between two endpoints. The RLNH must also supervise all endpoints that use a particular link and notify communicating endpoints when an endpoint becomes unavailable.

Connection Manager

The connection manager is responsible for providing reliable delivery of messages between endpoints. The purpose of the connection manager is to hide the details of the underlying media from the RLNH. If an endpoint should become unreachable the connection manager must notify the RLNH. In LINX there are several different connection managers, which are managed by the Connection Manager Control Layer. Examples of available connection managers are the Ethernet, TCP, Shared Memory and Point-To-Point connection managers which each handle a different type of connection media. Which connection manager that will be used for a connection is handled by the RLNH.

Every connection manager must regardless of which communication media it uses be able to deliver messages between endpoints according to the following set of rules [30]:

- Messages are delivered in the order they are sent.
- Messages can be of arbitrary size.
- Messages are never lost.
- The communication channel has infinite bandwidth.

If a connection manager is unable to follow these rules the connection must be reset and the RLNH must be notified.

Link Layer

The Link layer provides support for reliable communication across a variety of different communication media types. This layer includes platform specific operations for reliable communication across the communication media and therefore must be rewritten for each supported OS platform and communication media type.

Shared Memory Connection Manager

The shared memory connection manager provides efficient point-to-point communication on multicore architectures that support shared memory. The manager handles messages using a first come first served message queue and since the transportation media is considered reliable, all messages are received in sequential order. The link layer support that must be provided by the operating system for an application to be able to utilize shared memory is defined as a mailbox driver. The mailbox driver should apart from implementing the link layer part of the transmission protocol communicate with the shared memory connection manager using a set of callbacks. The full list of callbacks needed for such an implementation is provided in the "mailbox.h" file of the LINX source code [31]. Currently this driver is available for the Enea's OSE, but there is no driver available for other operating systems.

3.7 Summary

Currently maybe the most interesting feature in the design of a new message passing library is the compatibility with existing legacy code. Figure 3.4 shows an overview of how the libraries are related to the existing MPI industry standard. As mentioned in 3.3 a full MPI implementation is not suitable for embedded multicore systems due to its high memory overhead. Therefore the design should implement as much as possible of the MPI standard without introducing too much overhead. Of the implementations analyzed in this chapter, MSG provides the most extensive compatibility with the existing MPI standard. However as shown in [7] the MSG implementation significantly lacks performance in comparison to their MCAPI implementation which has more than 30 times lower latency for the smallest possible message size. Therefore, MCAPI should be considered over MSG when providing a standard library for embedded multicore.

The results from **??** would imply that the group operations of MSG might be redundant due to the lack of frequency in number of calls. However, the significantly higher frequency of blocking messages over non-blocking messages shows that blocking message calls is mandatory to provide compatibility with existing legacy code. In MCAPI the standard send function call, *mcapi_msg_send*, is a blocking function which further adds to this significance. LINX on the other hand only supports asynchronous messages which limits its current support towards legacy code.

Looking at the third requirement in section 3.2 that the message passing library must effectively provide a transparent abstraction of the underlying hardware to the developer. Although all of these implementations to some degree provide a good abstraction of the underlying hardware, MCAPI differentiates between connected and connectionless messages. Although this is a design decision that provides a significant performance increase for connected messages, it put the responsibility on the developer to choose between connectionless and connected communication between endpoints. A better alternative would be that the library itself could predict whether or not to use connected messaging. However predicting how frequent communication between two tasks is in an application is can be very difficult and create a lot of unnecessary overhead. The ideal solution would therefore be if the difference in performance between connectionless messages and connected messages would be insignificant. The implementation in this thesis will try to show if such a solution can exist. However, implementing a full scale library is not possible within the scope of the thesis. Hence Linx has been chosen to provide the high level part of a message library to show this. Linx however does not currently have any support for shared memory utilization between cores on Linux. Therefore Linx must be extended with a link layer driver that utilizes shared memory. How this can be achieved will be studied in the next chapter in further detail.

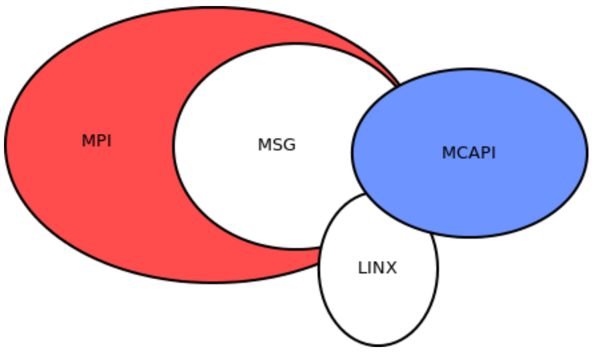


Figure 3.4: MPI Libraries

Chapter 4

Requirements on a New Message Passing Implementation

4.1 Introduction

This chapter will cover the requirements on the implementation of a new message passing API that utilizes the shared memory between cores on a multicore processor. Some interesting implementations that aims to solve this are presented, one of which will be suitable for implementation. The features that are required on a lightweight IPC mechanism between cores are the following:

Req_1: Low Memory Overhead	The message passing implementation must be able to scale well as the number of cores in the system increases. In particular the memory buffer allocation must be flexible so that the memory usage is kept at a minimum. The most basic requirement for scalable memory management is that each core only needs to maintain messaging buffers to the cores it is communicating with.
Req_2: Minimal Communication Overhead	The aim of the design is that the implementation should be nearly as lightweight as the connected messages of MCAPI. Thus the overhead introduced by the message header must be kept to a minimum.
Req_3: Built in User Space	The message passing implementation must be built entirely in user space. This is essential to minimize the usage of expensive system calls. Also legacy applications that use this programming model have been proven easy to migrate to new multicore architectures [32]. Ideally the solution should not need to involve the kernel when sending messages between endpoints. This to avoid the redundant copying where messages must be copied from user space to a shared memory region between endpoints.

4.2 Design 1: ZIMP: Zero-copy Inter-core Message Passing

ZIMP, which was presented in [33] is an inter-core communication mechanism that provides efficient zero-copy message passing that is able to outperform many existing solutions. Specifically on one-to-many communication calls ZIMP is able to vastly outperform other solutions. The solution uses a shared memory region where messages can be allocated and read directly. Within the shared memory region, communication channels can be created.

The size of the communication channels is determined by the number of messages and the maximal size of each message, which is passed as parameters to the create function for the communication channel. Each communication channel uses a circular buffer with an associated set of receivers. The communication channel can be written to by several senders and all associated receivers to that communication channel will be able to read messages written to the channel.

Associated with each channel is a variable called *next_send_entry* which indicates to the senders the next entry in the circular buffer where to write a message [33]. Each entry within the channel is associated with a bitmap. The bitmap is written by the sender and denotes which of the receiving processes should read that message. Each bit represents a receiving process and the bit i is set to one when the receiving process with identity i should read the message stored in that buffer. The receiver then sets that bit to zero whenever it reads the value from the buffer. In that way, the sending process can safely rewrite the position in the circular buffer when all the bits of the bitmap are set to zero.

The reading of the channel is managed by an array of variables, called the *next_read* array, where each variable corresponds to the index of the next value in the buffer that should be read by each process. Similarly to the bitmap associated with each buffer, the variable at position i in the array points to the next value to read for the i:th receiving process.

4.2.1 Suitability for Implementation

The implementation satisfies **Req_1** since the number of communication channels can be managed through parameters passed to the library. Furthermore, the bitmap associated with each communication channel gives the possibility to only use one communication channel for all messages of the system. This design also meets **Req_2** since the only information needed to pass a message is managed using the variables *next_send_entry, next_read* and the bitmap. The only downside to the message overhead of this implementation is that each message must be accompanied by the bitmap which is as large as the number of possible receivers. The designers themselves note that, in configurations using messages of size less than 1kB and only one receiver, ZIMP is outperformed by the performance of Barrelfish MP [33]. Interestingly though it is still able to outperform many traditional solutions such as Unix domain sockets and Pipes.

Looking at the third requirement, this solution is fully implementable in user space. However, each message must after being read by the receiver be copied into that process local memory space. This introduces some redundant message copying when there is only one sender and receiver.

4.3 Design 2: A New Approach to Zero-Copy Message Passing with Reversible Memory Allocation in Multi-Core Architectures

The approach presented in [34] provides a design of a message passing technique that can be managed operating entirely in a user space shared memory segment between processes. The main idea is to use a smart pointer design to address shared memory segments, which automatically is able to translate between the virtual address space of each logical process within the system. That way, only a pointer to where the message content is stored in memory needs to be copied between the two communicating processes.

4.3.1 Shared Memory Heap Structure

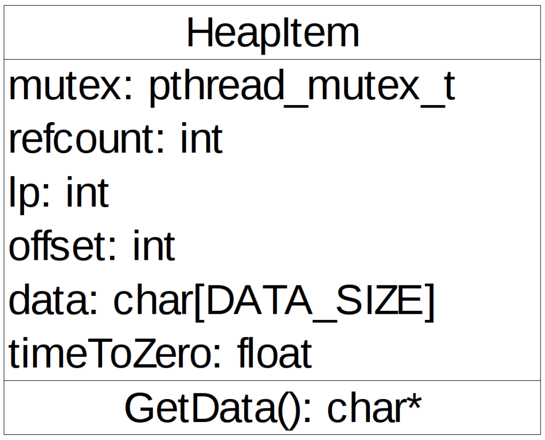


Figure 4.1: Shared Memory Heap Item

The design is implemented using user space shared memory segments. Within each segment, a shared heap object is allocated. An overview of the structure of a shared heap object is shown in Figure 4.1. The shared heap can contain segments of various sizes where each segment can be identified from the offset of how far from the beginning of the heap it is located. Therefore the only thing a process needs to be able to use the heap is to keep a pointer to the beginning of the heap.

4.3.2 Smart Pointer Design

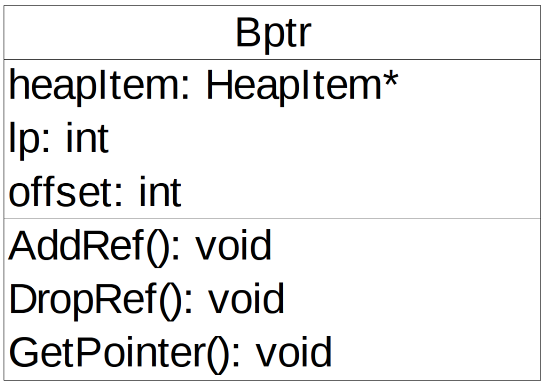


Figure 4.2: Smart Pointer Structure

The smart pointers is what really makes this approach work. Normal C-type pointers will not work when passed between different processes. This is due to that the virtual address to a shared memory segment for one process does not necessarily need to be the same as for another process. Therefore this design instead utilizes smart pointers which contain metadata about where the message is stored in memory. A UML-diagram of the smart pointers used in this approach is shown in Figure 4.2.

The pointers are passed between processes through shared memory where the pointer is copied into the receiving process address space. Using the metadata stored in each smart pointer, the receiving process is then able to translate the pointer into the correct virtual address in its own address space.

4.4 Suitability for Implementation

This approach is able to meet all requirements but **Req_1**, since each process needs to keep a pointer to each of the other processes shared memory heaps. This would not scale well when scaling up to thousands or millions of processes running in the same system since each process would need to keep thousands of pointers in memory. Thus, for this implementation to be able to meet **Req_1**, a slight redesign of the pointer management is needed. If however the allocation could be redesigned so that perhaps each message type has its own buffer, the amount of pointers each process needs to store could be reduced significantly.

4.5 Design 3: Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters

The design proposed in [35] proposes a design where messages of small size is handled differently from large messages. The motivation for this being that small messages are passed more frequently than large messages and therefore should be prioritized and handled more efficiently.

4.5.1 Design Overview

In this design, each process maintains a receive buffer for each other communicating process within the system. In this way, each buffer only contains messages from one sender. Furthermore each process also maintains a set of send queues to the other processes. The purpose of the send buffer is to allocate larger messages before they are read by the receiver. As an example, a set of 4 communicating processes would each maintain 6 buffers, three receive buffers and three send queues.

Small Message Transfer

When transferring small messages, the message contents are written directly into the receiving process' receive buffer by the sender. The receiving process then reads the message in its receive buffer and copies the message into its own memory.

Large Message Transfer

For large messages, it is no longer efficient to transfer the whole message contents directly into the receive buffer of the receiving process. Instead a control message is sent into the receive buffer that provides meta information on where in the sender's send queue that the message contents can be fetched. A typical message transfer can be described with the following steps [35]:

On the sender side:

1. The sending process fetches a free memory cell from its send buffer, copies the message from its source buffer to the free cell, and marks the cell as busy.
2. The process enqueues the cell into the corresponding send queue.
3. The process sends a control message containing the address information of the message contents by writing it into the receiving process' receive buffer.

On the receiver side:

1. The receiving process reads the control message from its receive buffer.
2. Using the information from the control message the process directly accesses the cell containing the message contents.
3. The process copies the message contents from the cell into its own destination buffer, and marks the cell as free.

Suitability for Implementation

The design is not able to meet **Req_1** due to that the amount of receive buffers needed by each process does not scale well as the number of communicating processes increases. Furthermore, once read, messages are copied into the local memory of the receiver. Since the design utilizes user space shared memory, the receiver should be able to keep the data in its receive buffer until it is no longer necessary. This causes the design to have a possibly unnecessary communication overhead, not satisfying **Req_2**.

4.6 Conclusion on Suitable Solution for Implementation

As derived from **??**, broadcast messages and other collective operations are an important part of a message passing library. However collective operations are not the most frequent type of communication in a standard system. Thus Design 1 is not really suitable due to the unnecessary overhead for point-to-point operations.

Comparing Design 2 to Design 3, the fact that Design 2 copies the message contents into its own memory is a clear disadvantage. Design 3 on the other hand fully utilizes the shared memory as message contents are kept in the shared memory region until no longer needed by the receiver. Overall Design 3 seems to be the design that best fit the requirements specified in section 4.1. However the amount of pointers to that each logical process must keep in memory to be able to access all shared heaps may cause large memory overhead for the implementation. A possible way to minimize this overhead could be to only use a limited amount of shared heaps that is shared between all logical processes. That way the amount of pointers that each process needs to keep in memory becomes more customizable.

Since all of these systems utilize user space shared memory, the message passing implementation cannot rely on the underlying operating system to provide error checking and debugability. Instead the system will have to provide error detection and traceability on its own.

Chapter 5

Providing Traceability in Embedded Systems

5.1 Introduction

In many domains of embedded systems such as the automotive domain, verification and certification of software is a requirement [13]. As the hardware and software complexity of embedded systems continue to grow, the verification process of system software is becoming increasingly difficult. The techniques that can be used when verifying software behaviour can either be techniques such as formal proof or built in tracing support within the software. This thesis aims to support the verification of embedded software by providing a built in execution trace in the message passing library implementation. This will not only be beneficial for software verification, but also to software debugging.

Debugging is often a difficult part of developing software for embedded systems and the same challenges apply to debugging as software verification that it is becoming increasingly difficult as hardware complexity keeps increasing. To simplify the debugging process it is therefore crucial to record enough information about a program's execution to easily be able to identify the source of errors and bugs. Built in tracing provides an efficient way to record information about program execution.

5.2 Tracing

Tracing is to store information about a specific state that has been reached within the application. Currently there are many available lightweight solutions that provide efficient tracing for embedded and general purpose systems such as Open Trace Format (OTF) [36] and the Linux Trace Toolkit next generation (LTTng) [37]. OTF provides both a specification standard and library for tracing on parallel platforms while LTTng is a complete tracing toolkit for the Linux operating system.

5.3 Tracing Goals

How much understanding of a program's execution that can be gained from an execution trace is entirely determined by how much information that is stored in the trace. Writing too much information into an execution trace however, can degrade the system performance greatly. This is especially true for embedded systems which have very limited main memory. Therefore it is important to make a good design choice of which information that should be stored within the trace. Furthermore the tracing information that is most relevant to the user may change during different stages of the application. For example more information about the system trace may be needed during testing and debugging than during runtime. Therefore the design of a good program execution trace should involve several different levels of tracing that each can be turned off or on individually. A good level design typically contains the following trace types:

5.3.1 Information Trace

The information trace contains information about the state of the system modules. This trace provides important information during system start up or shut down and provides information about which modules that were started/stopped correctly. During normal execution this trace may provide redundant information and thus should be turned off once the system start up has completed.

5.3.2 Error Trace

The error trace contains information about errors that occur within the system. Typically this trace is written to when a program exception occurs and is handled. Since this trace is only written to in the occurrence of system errors it does not affect the application performance and thus should never be turned off.

5.3.3 Debug Trace

The debug trace should provide detailed information of the program execution. From this trace a developer should be able to get the information about what methods the programs call and which local variables that are available along with their values. The amount of detail contained in this trace will generate huge amounts of traffic and therefore this trace should always be turned off during normal program execution.

5.3.4 Test Trace

Sometimes a program trace is needed to verify a certain test case. This is due to that in applications that consist of many connected modules it can be very difficult to read the output from a module directly. Instead a the module can generate a trace on which values that are received as input and which values that are sent as output. This is particularly useful in black box testing where the test trace provides the tester with an additional peephole into the module. Since test traces should be used only for verifying certain tests the trace should be turned off during normal execution.

5.4 Open Trace Format

OTF is a fast and efficient trace format library with special support for parallel I/O. It achieves good performance on single processor workstations as well as on massive parallel super computers [38]. Parallel I/O is important since each single trace generated using OTF is distributed into multiple streams. Each stream is represented as a separate file and may store events from one or more processes or threads. Each process however may only be mapped to one stream exclusively.

The traces generated by OTF are stored in ASCII format where each line represents one trace event. This makes the generated trace not only human readable but also easier to analyze by using existing tools like grep.

5.4.1 Open Trace Format 2 (OTF2)

In 2012 the Open Trace Format version 2 was announced which was a major redesign of the original Open Trace Format. The redesign was also based on experiences from the EPILOG trace format [39]. The OTF2 library is developed as part of the Score-P project [40] and is available under the BSD open source license. The version 2 design focuses on scalability and performance. One major change introduced was that the ASCII file format was replaced with a binary file format. Due to this a library for analyzing the generated trace is provided with the OTF2 implementation. This library is aimed to be more closely integrated into Scalasca [41] and Vampir [42], which are two major tools for trace analysis that utilized EPILOG trace format and OTF for trace generation respectively. This solves one of the major flaws of EPILOG and OTF that was the missing compatibility between the trace formats of Vampir and

Scalasca. Compatibility between the trace formats of the two tools are important since although both tools perform profiling analysis, they have different focus and may be able to spot different potential problems from the same execution trace.

OTF2 File Format

As with OTF version 1 each trace is distributed into multiple streams. The files of each trace is called an archive and are divided into four different file types [38]:

- Anchor file - A file with .otf2 suffix that contains meta data on archive organization.
- Global definition file - A file with .def suffix that stores global definitions that are equal for all locations.
- Local definition file - A file also with .def suffix that stores all definitions related to a specific location.
- Local trace file - A file directly related to the local definition file, with suffix .evt that stores all events that were recorded on the location related to the local definition file.

5.5 LTTng

Linux Trace Toolkit next generation (LTTng) is an open source technology for software tracing in Linux. It aims to provide highly efficient tracing tools for Linux.

5.5.1 LTTng Features

LTTng provides both a userspace tracer and a kernel tracer which comes along with many tools both for writing traces and reading them. The LTTV viewer is an advanced tool provided with LTTng that can analyze and show traces, both in text format and graphically [37].

The LTTng Userspace Tracker is designed specifically for providing detailed information about userspace activity [43]. It is designed to run entirely in userspace and hence the tracing module requires no operating system calls to operate. This feature is very fitting for providing lightweight tracing to applications designed to only operate in userspace.

5.6 Evaluation for Design

Both OTF2 and LTTng provide very interesting features. When compared side by side it is difficult based on only features alone to decide which trace to use over the other. However OTF2 has a benefit that is not shown in the features, which is that OTF2 is a defined standard. There is also much research material available for OTF2, which has proven difficult to find for LTTng, aside from the official documentation, during this evaluation.

From an evaluation standpoint, LTTng would be more interesting to use in an implementation since there currently seems to be much more development going on with the LTTng project. However, it may not be a reliable choice for an application due to the lack of academic content available.

Chapter 6

Designing a Standard Messaging API for Embedded Systems

6.1 Introduction

This chapter introduces the design proposed by this thesis. The design is built based on the requirements from the previous chapters and also how this adapts to the given use case. The main goal of the design is to become a bridge between the gap of MCAPI to MPI while still providing a message passing library that is suitable for embedded multicore. Therefore the library will provide compatibility with both the MPI and the MCAPI standards. Therefore the “e” in eMPI stands for embedded.

Several solutions that aims to make MPI more lightweight using MCAPI such as [44] do exist, however they require a full MPI implementation to be running on one of the system cores. The design proposed in this thesis uses a different approach and instead provides a more lightweight implementation of the MPI standard which is suitable for embedded multicore. The benefit of such a design would be to avoid the need for a full MPI implementation running on the system and thus making it more scalable down to embedded microcontrollers.

6.2 Software Requirements

This section will list the requirements that are going to be fulfilled by the implementation and the motivation why these features have been chosen.

sw_req_sync	The designed message passing library shall provide blocking, synchronous message passing calls.
sw_req_async	The designed message passing library shall provide non-blocking asyncronous message passing calls.
sw_req_mcapi_comp	The designed message passing library provides compatibility with the existing MCAPI standard.
sw_req_mpi_comp	For each API call there must be a corresponding function call in the MPI standard.
sw_req_seq_order	Messages must be received in the same order that they are sent.
sw_req_user_space	The message passing library must utilize the shared memory between cores through user space.
sw_req_ptr_scale	Each endpoint must only need to maintain a small amount of pointers as the number of processes grows large.
sw_req_msg_del	Reliable delivery is ensured by the library.
sw_req_trc_libinit	When the core library functionality has been initialized a trace containing information on available resources shall be generated.
sw_req_trc_libfinalize	When the core library functionality terminates successfully a trace containing information on available resources shall be generated.
sw_req_trc_register	When registering a new endpoint, the mailbox driver shall log the endpoint name along with a timestamp.
sw_req_trc_send	When sending a message, a trace containing the message contents and receiving endpoint shall be provided by the sender.
sw_req_trc_recv	When receiving a message, a trace containing the message contents and sending endpoint shall be provided by the sender.
sw_req_trc_snderr	If a message cannot be delivered to an endpoint, an error trace shall be generated by the message passing library.

6.2.1 Motivation for the Requirements

It was shown in [29] synchronous message calls occur even more frequently in MPI programs than asynchronous calls. Thus **sw_req_sync** is mandatory for any library. However, the use of asynchronous message calls in many applications can result in a significant performance boost over synchronous calls. Therefore the library must also satisfy **sw_req_async**.

To provide compatibility with both existing legacy code and future code it is important that the library must satisfy **sw_req_mcapi_comp** and **sw_req_mpi_comp** to provide compatibility with the two currently largest standards for message passing.

The library will provide an extension to the currently existing LINX implementation and therefore the library mailbox driver must satisfy the requirements of a connection manager for LINX. One of those requirements is that messages are delivered in sequential order. Since the library operates across a reliable media, i e. shared memory, there is no reason that messages should be received out of order by the receiver. Thus **sw_req_seq_order** can easily be satisfied and simplifies the usability of the library.

The implementation must provide efficient utilization of the shared memory between the cores in a multicore system. To provide efficiency, the library must be able to operate entirely in user space **sw_req_user_space**, to avoid unnecessary memory copies performed by the system kernel. It should be noted out that as mentioned in [CITE] the use of user space shared memory is only faster than regular shared memory techniques involving the kernel when communication is very frequent. The shared memory utilization must also be efficient in terms of memory footprint and therefore **sw_req_ptr_scale** must be satisfied for the library to scale well as the number of processes in the system grows large.

Since the communication will be operating entirely in user space, the implementation cannot rely on the kernel to provide any type of error tracing. Therefore the library must provide its own set of traces. These necessary traces that the library must provide are given by **sw_req_trc_libinit**, **sw_req_trc_libfinalize**, **sw_req_trc_register**, **sw_req_trc_send** and **sw_req_trc_recv**. Since the communication medium is considered reliable, no error traces need to be generated for the shared memory implementation. However the library still needs to provide a readable trace if a message cannot be delivered to an endpoint. This is covered by **sw_req_trc_snderr**.

6.3 System Overview

The system design can be divided into two main parts. The first one being the design of the MPI compatible application programming interface which will be described further in section 6.4. The second part consists of designing an efficient shared memory utilization implementation for the message passing library, which design is described in section 6.5.

Figure 6.1 shows an overview of the structure of the message passing library where the parts highlighted in red are the new features provided by this thesis. LINX will provide the base for the library. This will make the system easy to scale down to microprocessors. Furthermore LINX has a compatibility layer [45] with the OpenMCAPI library that makes it compatible with the MCAPI standard.

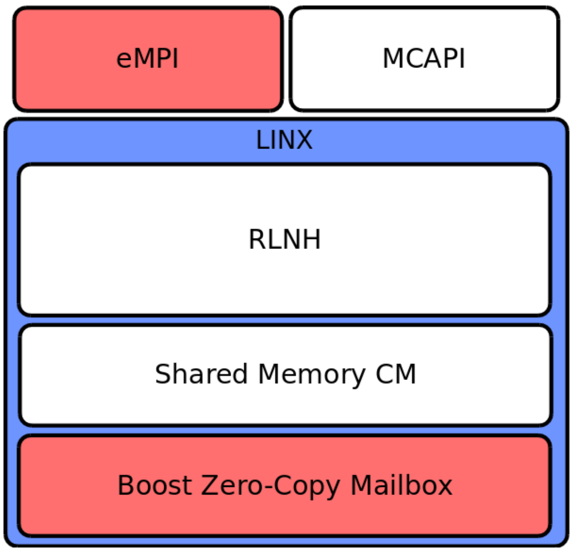
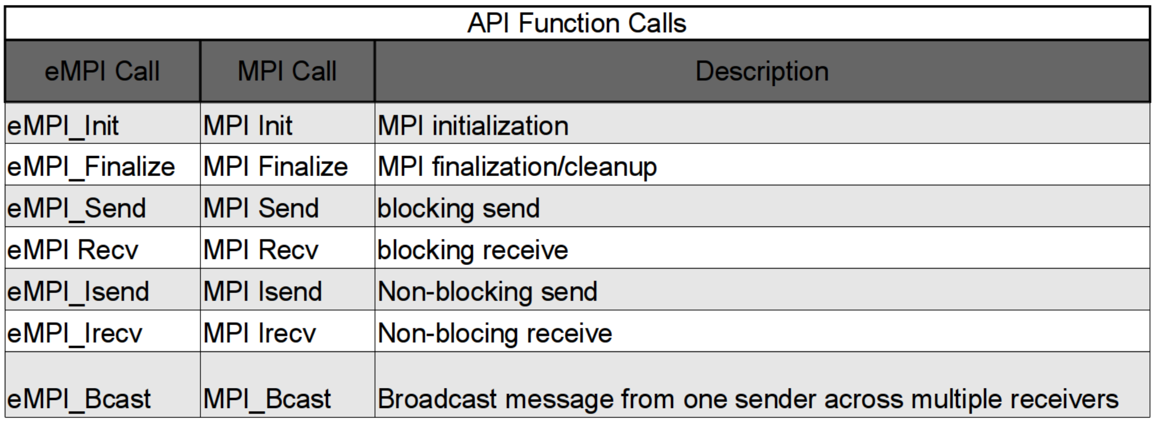


Figure 6.1: Design Architecture Overview

6.4 eMPI Application Programming Interface

The eMPI API will provide a subset of function calls from the MPI standard. A summary of all the function calls and their MPI counterpart is provided in figure 6.2. The selection of which MPI calls the API needed was largely based on the results from **??**. Furthermore MCAPI supports all of these operations apart from the *MPI_Bcast* call. The reason for adding this call is to provide more extensive support for collective operations to be built on top of eMPI. Since the broadcast message is the only true one-to-many operation, it is also the only

The support for collective operations will not be provided due to that in **??** the amount of calls where much lower than point-to-point operations. Also there is no support for collective operations in MCAPI, hence it is sensible to exclude support for collective operations in an implementation that supports both standards. Collective operations can however be implemented on top of both eMPI and MCAPI. To support this eMPI will provide support for broadcast messages. The reason being that Zero-Copy techniques can by their very nature provide scalable support for broadcast messages and hence implementing it on top of eMPI would be wasteful. The other collective operations from **??** can be implemented on top of eMPI without significant performance losses.



API Function Calls		
eMPI Call	**MPI Call**	**Description**
eMPI_Init	MPI Init	MPI initialization
eMPI_Finalize	MPI Finalize	MPI finalization/cleanup
eMPI_Send	MPI Send	blocking send
eMPI Recv	MPI Recv	blocking receive
eMPI_Isend	MPI Isend	Non-blocking send
eMPI_Irecv	MPI Irecv	Non-blocing receive
eMPI_Bcast	MPI_Bcast	Broadcast message from one sender across multiple receivers

Figure 6.2: eMPI Application Programming Interface

6.4.1 Built in Information and Debug Trace

The eMPI library will be accompanied with a built in trace mechanism to simplify the debugging and verification of applications that utilize eMPI. The traces provided at the API level will cover detailed information traces about the library initialization and tear down process. Also there must be a more detailed debug trace provided that gives full information about the details of the API calls performed and their arguments.

6.5 Boost Shared Memory Mailbox

The shared memory mailbox will act as a driver allowing the Shared Memory Connection Manager of LINX to utilize Zero-Copy shared memory communication. To achieve this the implementation will use Boost Interprocess [46], a C++ library that simplifies the use of common interprocess communication and synchronization mechanisms.

6.5.1 Boost Interprocess

Boost C++ libraries [47] is a collection of libraries that work well with the C++ standard library. All libraries are available under the Boost Software License which makes them available for both commercial and non-commercial use.

Boost interprocess is a library that provides a simple interface for creating and managing shared memory. The shared memory provided by boost is implemented through memory mapped files which are mapped into the address space of several processes so that it may be read and written to without using any operating system functions [48]. For this reason, the Boost interprocess library was used in the implementation to allocate shared memory segments.

6.5.2 Mailbox Design

Following the design of [34] the mailbox driver will utilize shared memory heaps and smart pointers to send data between processes. The structure of the shared heap objects and the design of the smart pointer will be the same as described in 4.3. The driver will be highly configurable in that the amount of shared heaps allocated between processes can be anything between a single heap or multiple heaps. Also a processes is not limited to have one mailbox for any type of message, but may have one mailbox for each message type, or sender id.

6.5.3 Built in Debug and Error Trace

Since the mailbox driver will be operating entirely in user space, the library cannot rely on any error handling from the operating system. Therefore the library must itself provide an efficient way to detect and handle errors. Furthermore additional, more detailed traces must be provided to simplify debugging.

Chapter 7

Implementation of Zero-Copy Communication on Linux

7.1 Introduction

This chapter will describe what the implementation covered and its limitations. The goal of the implementation is to demonstrate some of the key features of the design. Specifically the performance of the zero-copy mailbox driver needs to be investigated and evaluated. Hence the main focus of the implementation will be to evaluate whether the mailbox driver provides enough performance to be worth integrating into LINX.

7.2 Delimitations on the Implementation

Due to the time restrictions of this thesis, some of the desired features of the design cannot be implemented and are left to future work. The main part of the design that is being left out is the designed API which was described in section 6.4. The reason why this part of the design is less prioritized than the mailbox driver is due to that without an efficient link layer implementation, no interesting results can be obtained from it. There is no need to prove whether or not LINX can be made more MPI-compatible and there is no need to evaluate the performance of a subset of MPI, since that has already been done in [9]. The truly interesting evaluation to be made on the eMPI API is its performance against the scalar channels and connectionless messages of MCAPI. However this requires the link layer to efficiently utilize shared memory.

7.3 Tracing With OTF2

As described in section section 5.6, there seems to be much more content on the performance of OTF2 than LTTng. Therefore OTF2 has been selected for use with this message passing library. In the basic implementation provided by this thesis, the trace will only cover error messages. However in for the implementation to be usable within the industry, the trace must also be extended with a more detailed debug trace. The error tracing added to the basic implementation is a bare minimum for an implementation that operates entirely in user space and should not affect performance at all during normal operation.

7.4 Implementation Overview

Since the boost C++ library was used to manage the shared memory of the library, the core of the library was implemented using C++. As such the core implementation was implemented in a object oriented fashion. Figure 7.1 shows a UML class diagram of the library core. The core was then linked to to a C API through the function calls provided by the mailbox_manager class. This part of the implementation could also have been implemented in C in a similar fashion using mmap.

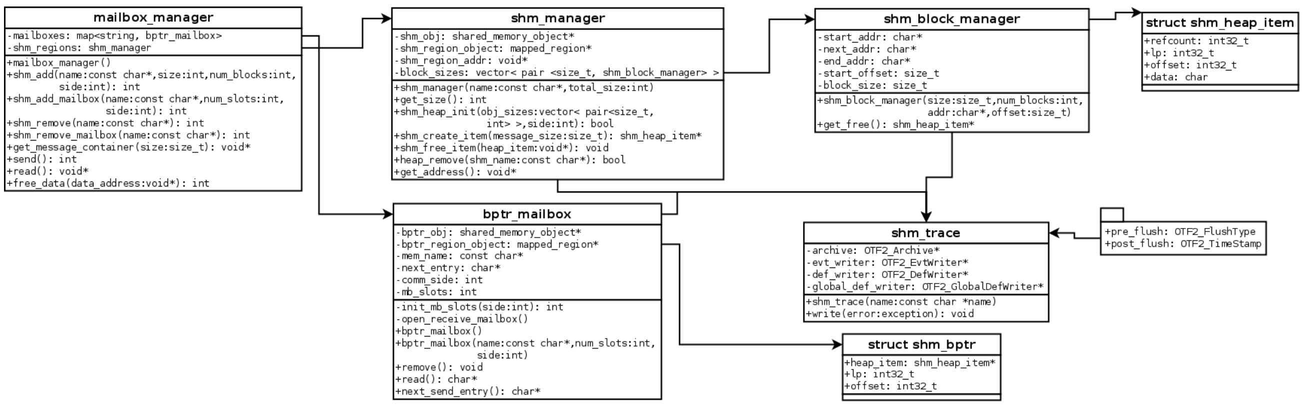

mailbox_manager
-mailboxes: map<string, bptr_mailbox>
-shm_regions: shm_manager
+mailbox_manager()
+shm_add(name:const char*,size:int,num_blocks:int, side:int): int
+shm_add_mailbox(name:const char*,num_slots:int, side:int): int
+shm_remove(name:const char*): int
+shm_remove_mailbox(name:const char*): int
+get_message_container(size:size_t): void*
+send(): int
+read(): void*
+free_data(data_address:void*): int
shm_manager
-shm_obj: shared_memory_object*
-shm_region_object: mapped_region*
-shm_region_addr: void*
-block_sizes: vector< pair <size_t, shm_block_manager> >
+shm_manager(name:const char*,total_size:int)
+get_size(): int
+shm_heap_init(obj_sizes:vector< pair<size_t, int> >,side:int): bool
+shm_create_item(message_size:size_t): shm_heap_item*
+shm_free_item(heap_item:void*): void
+heap_remove(shm_name:const char*): bool
+get_address(): void*
shm_block_manager
-start_addr: char*
-next_addr: char*
-end_addr: char*
-start_offset: size_t
-block_size: size_t
+shm_block_manager(size:size_t,num_blocks:int, addr:char*,offset:size_t)
+get_free(): shm_heap_item*
struct shm_heap_item
+refcount: int32_t
+lp: int32_t
+offset: int32_t
+data: char
bptr_mailbox
-bptr_obj: shared_memory_object*
-bptr_region_object: mapped_region*
-mem_name: const char*
-next_entry: char*
-comm_side: int
-mb_slots: int
-init_mb_slots(side:int): int
-open_receive_mailbox()
+bptr_mailbox()
+bptr_mailbox(name:const char*,num_slots:int, side:int)
+remove(): void
+read(): char*
+next_send_entry(): char*
shm_trace
-archive: OTF2_Archive*
-evt_writer: OTF2_EvtWriter*
-def_writer: OTF2_DefWriter*
-global_def_writer: OTF2_GlobalDefWriter*
+shm_trace(name:const char *name)
+write(error:exception): void
+pre_flush: OTF2_FlushType
+post_flush: OTF2_TimeStamp
struct shm_bptr
+heap_item: shm_heap_item*
+lp: int32_t
+offset: int32_t

Figure 7.1: UML Diagram of Mailbox Driver

Chapter 8

Use Case Implementation: Online Face Recognition

8.1 Introduction

To demonstrate the efficiency of the solution presented in this thesis, it will be implemented into a real world application. The application is managing real-time face detection in video. This use case for the application is very suitable to demonstrate the efficiency of the message passing library. This is due to that in many video applications, the communication is responsible for more than 45% of the overhead of the application [49]. The application will be a combined effort from 3 master thesis students. The zero-copy implementation provided in this thesis will be used for communication between the system components. Furthermore the work of another student will provide the application with runtime adaptation of the program behavior. The third students contribution will be to parallelize the application using various automatic tools. This is essential for optimal performance of the application on multicore systems.

8.2 OpenCV

The Open Source Computer Vision Library (OpenCV) is an open source library for image analysis applications. It was originally released by Intel and is available under the BSD license [50]. In this implementation, OpenCV was used to read and modify image data through its C programming interface. The built in functionality for face detection however was not used. Instead one of the students in the team implemented both a training application and a facial recognition application. These two applications were written as sequential programs that were later parallelized using various tools for automatic parallelization. The functionality of these two applications within the implemented face recognition system along with the other system components will be described in further detail in the next section.

8.3 Application Overview

The application can be divided into four main components. Each of which are described in further detail in the following subsections. In the original design of the application, there was also a fifth component where the identities of detected faces in the video was identified. This module was however removed due to lack of time, but could however easily be added in afterwards. The addition of that fifth module would probably have been able to further demonstrate the efficiency of the zero-copy communication.

8.3.1 Video Reader

The video reader module will be able to read video frames from a video stream and send the frames to the Face Recognition Classifier module. The frames will be sent as Iplimages which is an image format provided in OpenCV.

8.3.2 Face Recognition Trainer

This part of the application takes a large number of images, both images containing faces and images that do not contain faces as training data to produce a classifier. Once completed the classifier is written to an xml file that is readable by the Face Recognition Classifier.

8.3.3 Face Recognition Classifier

In this module, images from the video reader are received and analyzed to detect faces within them.

8.3.4 Video Output Module

This module receives the image frame that has been modified by the Face Recognition Classifier and outputs it onto a screen. The modified image will contain squares drawn around any detect faces within the image frame.

8.4 Zero-Copy Integration

This section will describe how the zero-copy message passing library was used within the face recognition module. To utilize the zero-copy mechanism to its fullest, the communication between modules is designed in such a way that actual image data is never copied between modules. Instead, only smart pointers are passed between the various modules. This means that for each video frame, of size 300kB, that is processed through the application, only 2 smart pointers, i. e. 30 bytes of data needs to be copied between the modules. The flow of data through each stage of the application is illustrated in figures 8.1-8.3. The large size of each data frame also indicates another reason why using zero-copy is a better design choice than many of the alternatives. For instance sockets have a maximum data size that is limited by the size 64kB. This means that using sockets, not only would the data need to be copied multiple times throughout the system, but also the data would have to be fragmented. This would cause the communication to be many times slower using sockets.

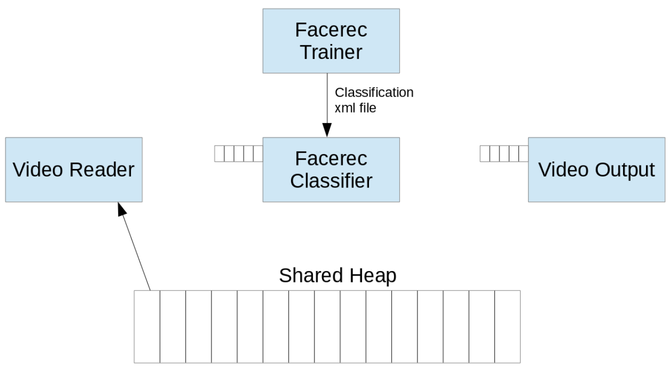


Figure 8.1: The video reader requests a message container on the shared heap. Reads a frame from the video stream and stores it directly into the shared heap.

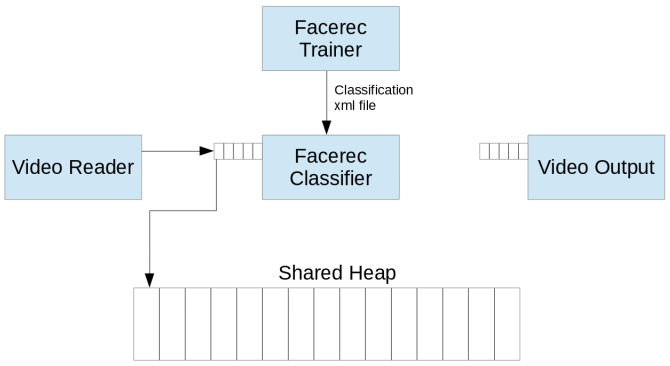


Figure 8.2: The video reader sends a smart pointer to the face recognition classifier, which directly reads the image data from the shared heap. The image is then processed and any faces within the image are detected.

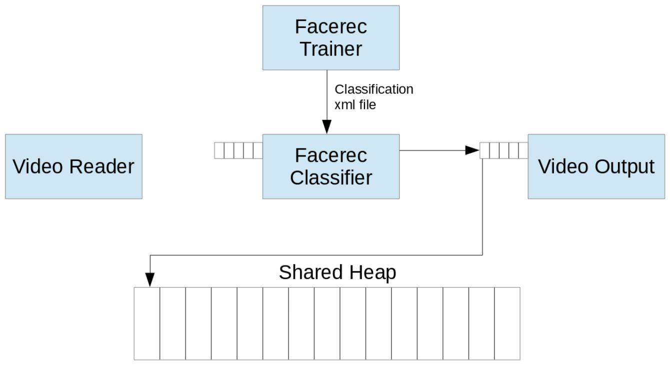


Figure 8.3: The face recognition software sends a smart pointer to the video output module which outputs the processed image to a screen.

Chapter 9

Evaluation

9.1 Introduction

This chapter presents how the implementation was evaluated, and the obtained results from that evaluation. The goal of the evaluation is to obtain more knowledge about the behavior of the implementation, and through the results understand more of its advantages and disadvantages. Furthermore, it is crucial to obtain results where the communication speed of the zero-copy implementation is compared to the old LINX implementation using sockets. The zero-copy implementation is expected to be significantly faster than sockets overall. However, it would also be interesting to see whether or not there are situations where sockets are faster.

9.2 Zero-Copy Implementation Evaluation

9.2.1 Evaluation Of the Design

Providing a true zero-copy implementation for message passing has given a clear view of both the pros and cons of such a solution. The main benefit of a true zero-copy solution is clearly that the number of times data is copied through the system is kept to a minimum. The integration of the zero-copy implementation in the use case application, described in the previous section clearly showed this, where the same large message was passed between multiple processes without using a single memory copy once allocated into shared memory.

The cons of using a true zero-copy implementation are that it is difficult to fully abstract the underlying message passing implementation from the user. Since the message data that the receiver of a message will read will remain in shared memory, the implementation must provide a way for data to be marked as free once the receiver is done reading the data. This can be solved either by providing a mechanism for the receiver to free the data once read, or by having the library perform its own garbage collection of shared data. A completely different solution that only works for point-to-point communication between two processes is the solution presented in [49], where the read message is copied into the receivers local memory. This causes an additional memory copy for each message, but has the benefit that the shared data can be freed immediately once read.

When it comes to robustness, the zero-copy design has one major disadvantage. The shared memory regions are allocated in memory as memory mapped files. This means that the shared memory regions are kernel persistent and thus will remain in memory until explicitly deallocated or a system reboot. This is fine during normal program behavior, where the library can simply deallocate shared memory when it is no longer needed. In unexpected program behavior, however, this becomes difficult since in the case of an unexpected program crash the program may never reach the point where the shared memory will be deallocated. This problem can however be solved by maintaining link supervision on both sides of the communication. In that way, if one endpoint in the communication would crash unexpectedly the other endpoint can deallocate the memory shared between the two.

9.2.2 Performance Evaluation Setup

In an application where communication is very frequent, the most important characteristic of the implementation is how high the end-to-end latency is. Therefore the initial performance testing focused on measuring the end-to-end latency. The overall latency of the zero-copy implementation was evaluated through sending messages of various sizes between a sender and a receiver in a simple producer consumer fashion. The implementation is mainly targeted at passing standard C type structs between endpoints and the data transferred in the evaluation was the simplest of structs, namely character strings. In each test, the average transfer time of sending 10 messages of the same size was measured.

The testing was done on two different machines, one using an Intel Pentium 4 processor at 3.0GHz with 1M L2 cache and 2GB of RAM. The second test machine was equipped with an Intel Core 2 Quad Q9550 at 2.83GHz with 12MB L2 cache and 4GB of RAM.

9.2.3 Initial Prototype Testing

To protect the transferred data, once sent, from further modification by the sender, the message data was copied into shared memory. This would cause the implementation to not become a true zero-copy implementation, but would provide a protective layer between sender and shared memory. Figure 9.1 shows the result from the initial tests. It can be derived from the figure that the transfer time increases almost linearly as message size grows larger. This initial result is not really desirable for a zero-copy implementation, since the point-to-point latency to pass a smart pointer from sender to receiver, i.e., sending a message, should ideally be the same regardless of message size. In the next section it will be shown that removing the additional copy from sender to shared memory removes this undesirable performance.

What can also be derived from Figure 9.1 is a result that was rather expected. Since the shared memory is implemented using memory mapped files, the minimal size that can be allocated is equal to one page size, i.e., 4kB. Between the message sizes 0 and 4kB the transfer time is almost constant. However between 4kB and 8kB the transfer time more than doubles, from 10 msec to 20.5 msec.

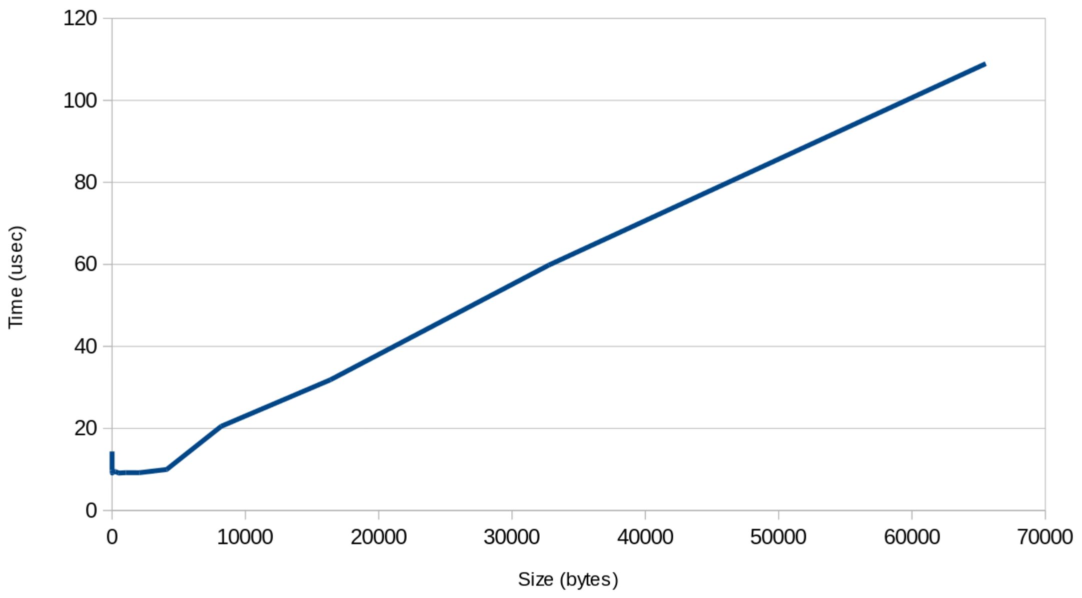


Figure 9.1: Performance of the initial library prototype

9.2.4 Performance Evaluation of Final Implementation

Due to the undesirable performance discovered during the initial testing described in the previous section, the message passing was redesigned so that no memory copies occurred between the sender and the receiver. Instead, the sender requests from the message passing library a container where it can store the message contents. In these tests the end-to-end latency was measured through the transfer time when sending a smart pointer from the sender to the point where the receiver has obtained a translated pointer in its own address space. The results of these tests are shown in Figure 9.2 and Figure 9.3. The results obtained are much more desirable than the initial implementation provided. On the Pentium 4 machine, even though multiple measurements per message size were being used, there is a slight noticeable increase in latency as message size grows larger. However the difference between the lowest measured latency 9.2 microseconds achieved for a message size of 256 bytes, and the highest measured latency 15.8 for a message size of 500kB, seems to indicate that the message size has a very small impact on the overall transfer time. This becomes even more clear when looking at the results of the Core2 machine, which has much better multithreaded performance than the Pentium 4. Here the difference in measured latency is less than one microsecond between the smallest message size of 2 bytes and the largest message size of 512kB. Hence the desired results where the latency is not affected significantly by message size is achieved.

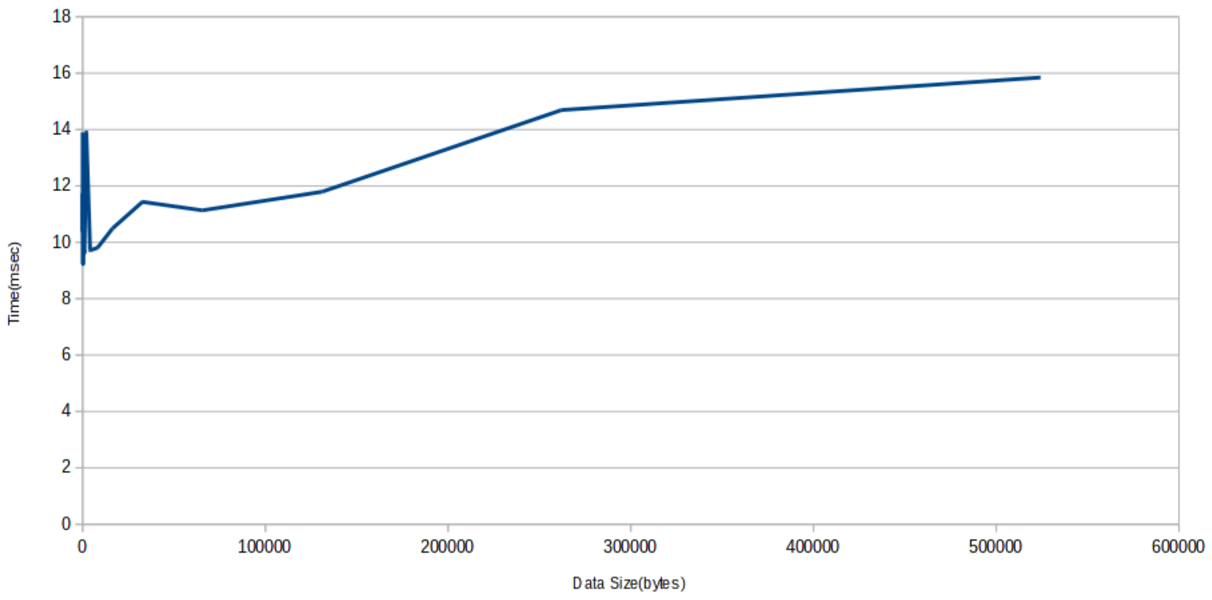


Figure 9.2: Performance test on an Intel Pentium 4

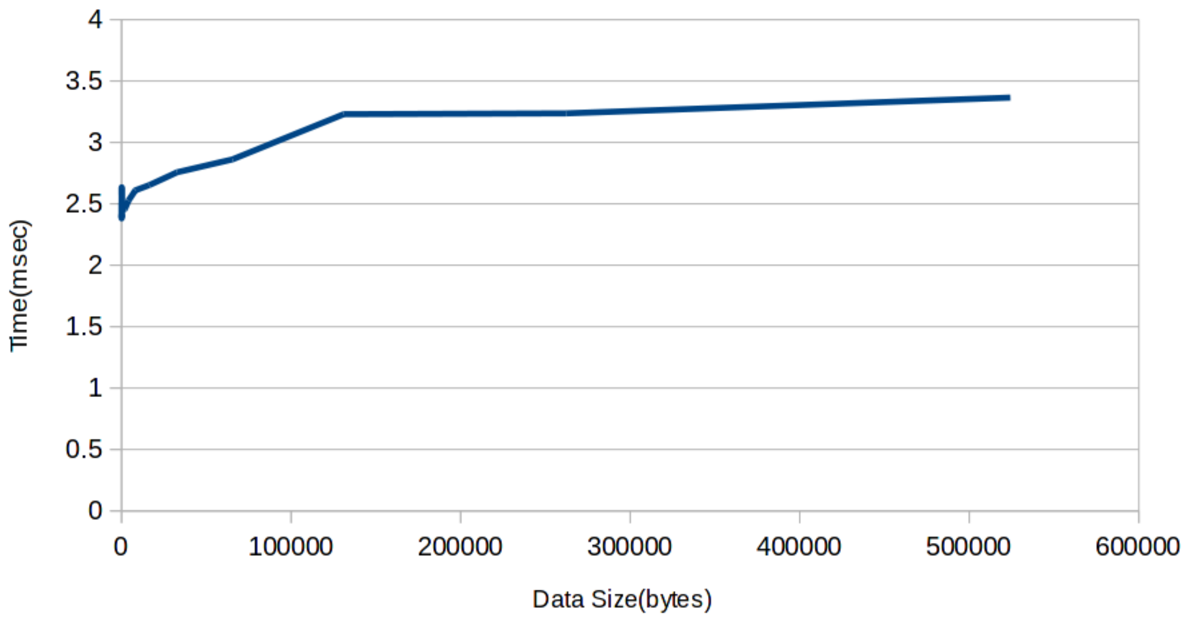


Figure 9.3: Performance test on an Intel Core2 Quad

9.3 LINX Integration Evaluation

This section evaluates the performance of the zero-copy implementation with the perspective whether or not it would be worth integrating into LINX, and what benefits and trade-offs such an integration would provide.

9.3.1 Performance Against Existing LINX Implementation

The current LINX implementation uses sockets as communication medium on Linux. Therefore there is not really any valuable point in comparing the existing LINX implementation to the zero-copy implementation. In this situation where shared memory is present on the test machine, memory mapped should always be faster than sockets for any reasonable amount of messaging data. However, it may be interesting to know how much performance can potentially be gained from integrating the zero-copy implementation into LINX.

In this test the provided benchmark program with the LINX distribution was used and configured so that it measures the execution time for sending 10000 messages of each size. The test was run multiple times on the Intel Pentium 4 machine and the average transfer time of 10 consecutive runs was used. To compare the results with the zero-copy implementation, a test program for the zero-copy implementation that measures execution time across the same amount of messages was written. The source code for this program can be found in Appendix A and the results of these tests can be found in Figure 9.4.

As can be seen from the results, the zero-copy implementation runs significantly faster. These results clearly indicate that integrating the zero-copy would be of interest. The addition of LINX on top of the zero-copy implementation will, however, introduce some extra overhead. However, the zero-copy implementation is almost 100 times faster for small messages and about 5 times faster for the largest message size which is a very significant difference.

The LINX benchmarking application does not run on message sizes above 64kB, since sockets are limited to that size on a single packet. This means that for messages larger than 64kB the current LINX implementation would have to fragment the data, which probably would cause a significant slowdown.

Therefore the zero-copy implementation should be much faster above that limit, since it does not need to fragment data packets for sizes larger than 64kB.

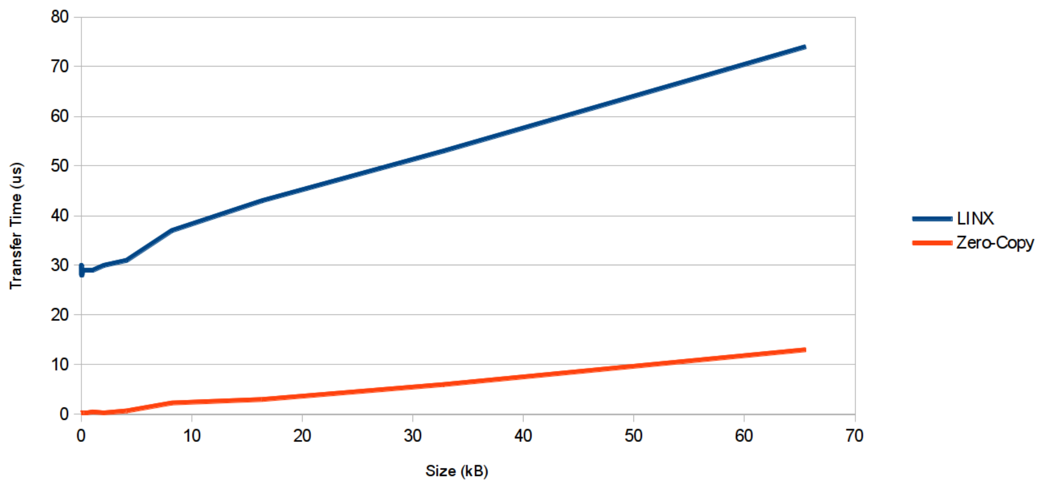


Figure 9.4: Performance Comparison Against LINX

9.3.2 Necessity of LINX

Since the zero-copy implementation uses shared memory, which is considered a reliable communication medium, an additional abstraction layer such as LINX for reliable point-to-point communication might seem unnecessary. However, LINX provides link supervision which is very important for system stability when dealing with kernel persistent memory mappings. Therefore if one endpoint in the communication would crash unexpectedly, the mapped data can be freed from memory by the other endpoint. Without link supervision the memory mapped by the application would remain in memory until a system reboot, which in embedded applications is not an option.

Another very crucial point to note is that LINX is a library that is already widely deployed within embedded systems and as mentioned in section 1.1, the industry is currently shifting from decentralized systems to multicore. Therefore the implementation provided in this thesis is not that valuable to the industry in itself. Instead by being integrated into LINX, existing applications that use LINX can start utilizing the zero-copy performance immediately without changing a single line of code. The only needed change necessary to start using the zero-copy implementation through LINX is the LINX installation itself. This is much more beneficial than needing to change the actual source code to start using a new library. In fact, in some cases part of the source code might not be available and in those cases changing the source code is not even an option.

9.4 Conclusion

Through the evaluation it was shown that the message size is not affected by message size, which is a very important feature when it comes to the overall performance of the application. Furthermore it was shown that there is a significant performance increase that can be gained from adding the zero-copy functionality into LINX.

Chapter 10

Conclusions and Future Work

10.1 Thesis Contributions

Following up from the problem statement presented in section 1.2, the main goal of this thesis was to design and implement a high performance IPC mechanism satisfying the following requirements:

Efficient Utilization Of Shared Memory Between Cores

The implementation provided in this thesis shows that zero-copy based message passing provides highly efficient communication by utilizing the shared memory between cores. It is able to achieve much higher performance than many of the existing IPC mechanisms like sockets.

Low Communication Overhead

By design, zero-copy achieves a minimal communication overhead since there is no copying of the actual message data needed to pass information. Furthermore, no expensive operating system calls are needed for communication once shared memory has been allocated.

Ease Of Use

The designed message passing library provides a simple interface with multiple layers, providing a high level abstraction of the underlying hardware for the application developer. Even on the low level of the zero-copy implementation, much of the complexity of the shared memory structure are abstracted away from the developer.

The layered design also lowers the amount of work required when moving to a new hardware platform. Since only lower layers of the message passing library needs to change when porting the library to a new platform, no changes to the application are needed.

10.2 Conclusions

Zero-copy communication certainly has some advantages over other communication mechanisms when it comes to sending large messages. Overall when it comes to performance of communication heavy applications, zero-copy seems to be the obvious choice. It is however limited by that the minimum possible message size is 4kB, i.e. one page size. This makes it potentially slower than conventional IO communication in applications where communication is very infrequent.

An important requirement for any zero-copy implementation is that it must be formally verified, either by static analysis or extensive testing. This is due to that the sender of messages is writing directly into the shared memory and may invalidate the shared memory structure. In many cases throughout the implementation in this thesis it has been difficult to determine whether or not an occurring software bug has been caused by the application using the library or is a bug within the library itself.

Memory mapped files are also limited by the maximum amount of what is addressable by the host system. This however seems to be a rather small drawback in the current application since about 256MB of data could be addressed during the tests on the Pentium 4 test system with 2GB of RAM and about 512MB on the Core2 Quad test system with 4GB of RAM. This limitation could also be avoided by extending the current implementation to utilize multiple mapped files.

Right now MCAPI seems to be the most adaptable and suitable library for on chip communication in multicore systems. However the current state of the upcoming standard seems to be that it is limited by that the connectionless messages are not as flexible as other solutions like MPI. Also there seems to be a lot of room for alternative solutions to inter-core communication since it is a rather new subject that has only been around for a few years.

10.3 Future Work

The current version of the zero-copy implementation only supports mapping a single memory mapped file, but could as mentioned in the previous section be extended to utilize multiple mapped files. This would allow the implementation to utilize more than the maximum amount of memory that can be mapped into a single file. However potentially this extension might run into another barrier instead, which is how much memory that can be mapped to into a single process address space.

One important aspect of the zero-copy IPC mechanism that has not been covered in this thesis is polling overhead for the receiving endpoints. Since there are no operating system calls involved when reading from a mailbox, the receiver cannot wait for a kernel interrupt to occur when trying to receive a message. Instead, the receiver must continually poll the mailbox for data. This potentially could have a huge performance impact.

LINX provides a compatibility layer with the MCAPI standard which makes MCAPI able to utilize LINX for connectionless messages. Since MCAPI currently is more limited than MPI and sockets when it comes to connectionless messages, testing the performance of the zero-copy message passing for MCAPI would be an interesting comparison. Especially a comparison between connectionless messages using LINX and the scalar channels of MCAPI would be interesting. If zero-copy messaging turns out to have similar performance to scalar channels it may be a viable alternative to those with the added benefit of more flexible message size.

Appendix A

Initial Benchmark Program

```
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <iostream>
#include <fstream>
#include <ctime>
#include <sys/time.h>

#include ”mailbox_man.hpp”
using namespace std;

timespec ts1;
timespec ts2;
ofstream out_file;

int i;

int main(int argc, char *argv[])
{
  // Setup
  mailbox_man sender;
  mailbox_man receiver;


  out_file.open(”transfer_time.csv”);
  for (int i = 1; i < 678435456; i=i*2)
  {
    // Do setup
    sender.shm_add(”Benchmark”, i, 1, 1);
    sender.shm_add_mailbox(”ReceiverMb”, 1, 1);
    receiver.shm_add(”Benchmark”, i, 1, 0);
    receiver.shm_add_mailbox(”ReceiverMb”, 1, 0);
    // Output size
    out_file << i << ”,␣”;


    char *message = (char*) sender.get_message_container(i);
    sprintf(message, string(i, ’a’).c_str());
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ts1); // Works on Linux
    sender.send(”ReceiverMb”, (void*) message);
    const char *read_message = (char*) receiver.read(”ReceiverMb”);
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ts2); // Works on Linux

    // out_file << (float) clk1 << endl;
    out_file << ((&ts2)−>tv_nsec − (&ts1)−>tv_nsec) / 1000 << endl;
    sender.shm_remove(”Benchmark”);
    receiver.shm_remove_mailbox(”ReceiverMb”);
  }

  out_file.close();
  out_file.open("setup_time.csv");

  return 0;
}
```

Comparison Benchmark for Zero-Copy vs. LINX

```
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <iostream>
#include <fstream>
#include <ctime>
#include <sys/time.h>
#include <pthread.h>

#include "mailbox_man.hpp"
using namespace std;

void *read_data(void *message_size) {
    mailbox_man receiver;
  size_t size = (size_t) message_size;

    receiver.shm_add("ZeroComparison", size, 1000, 0);
    receiver.shm_add_mailbox("ReceiversMailbox", 1000, 0);

    for (int i = 0; i < 1000; ++i)
    {
      void* message = NULL;
      while(message == NULL) {
        message = receiver.read("ReceiversMailbox");
      }
      char *data = (char*) message;
      // modify some of the data
      data[0] = 'c';
    }
    receiver.shm_remove_mailbox("ReceiversMailbox");
  receiver.shm_remove("ZeroComparison");
  pthread_exit(NULL);
}


int main(int argc, char const *argv[])
{
  size_t message_size = 16348;
  mailbox_man sender;
    pthread_t receiver;
    timespec ts1;
  timespec ts2;
  void *status;

  if (argc == 2)
  {
    printf("Parsing␣input␣message␣size\n");
    message_size = atoi(argv[1]);
  }

  sender.shm_add("ZeroComparison", message_size, 1000, 1);
    sender.shm_add_mailbox("ReceiversMailbox", 1000, 1);
    pthread_create(&receiver, NULL, read_data, (void *) message_size);

    clock_gettime(CLOCK_REALTIME, &ts1);
  for (int i = 0; i < 1000; ++i)
  {
    void *message = NULL;
    while(message == NULL) {
      message = sender.get_message_container(message_size);
    }
```

```
      memset(message, 1, message_size);
      sender.send("ReceiversMailbox", (void*) message);
    }
    // Wait for receiver to finish
    pthread_join(receiver, &status);
      clock_gettime(CLOCK_REALTIME, &ts2);
      cout << ((&ts2)->tv_nsec - (&ts1)->tv_nsec) / 1000 << endl;
    sender.shm_remove_mailbox("ReceiversMailbox");
    sender.shm_remove("ZeroComparison");
    pthread_exit(NULL);
}
```

Bibliography

[1] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

[2] `http://www.enea.com/linx`. [Online; accessed 20140206].

[3] `http://www.crafters-project.org/`. [Online; accessed 20140206].

[4] `http://www.artemis-ia.eu/`. [Online; accessed 20140206].

[5] Vishnuvardhan Avula. Adapting enea ose to tilepro64 - designing linx for high performance inter-core process communication targeting many-cores. July 2013.

[6] Pierre-Louis Aublin, Sonia Ben Mokhtar, Gilles Muller, and Vivien Quéma. REICoM: Robust and Efficient Inter-core Communications on Manycore Machines. Technical report, INRIA Rhône-Alpes, 2012.

[7] L. Matilainen, E. Salminen, T.D. Hamalainen, and M. Hannikainen. Multicore communications api (mcapi) implementation on an fpga multiprocessor. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 286–293, July 2011.

[8] James Psota and Anant Agarwal. rmpi: Message passing on multicore processors with on-chip interconnect. In *Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC'08, pages 22–37, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] Shih-Hao Hung, Wen-Long Yang, and Chia-Heng Tu. Designing and implementing a portable, efficient inter-core communication scheme for embedded multicore platforms. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2010 IEEE 16th International Conference on*, pages 303–308, Aug 2010.

[10] `http://www.uppaal.org/`. [Online; accessed 20140206].

[11] Git. `http://git-scm.com/`. [Online; accessed 20140522].

[12] Jenkins ci. `http://jenkins-ci.org/`. [Online; accessed 20140522].

[13] I. Gray and N.C. Audsley. Challenges in software development for multicore system-on-chip development. In *Rapid System Prototyping (RSP), 2012 23rd IEEE International Symposium on*, pages 115–121, Oct 2012.

[14] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, April 1979.

[15] W.R. Stevens. *UNIX Network Programming: Interprocess communications.* The Unix Networking Reference Series , Vol 2. Prentice Hall PTR, 1998.

[16] Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra Marathe, and Mark Moir. Message passing or shared memory: Evaluating the delegation abstraction for multicores. In Roberto Baldoni, Nicolas Nisse, and Maarten Steen, editors, *Principles of Distributed Systems*, volume 8304 of *Lecture Notes in Computer Science*, pages 83–97. Springer International Publishing, 2013.

[17] A.C. Sodan. Message-passing and shared-data programming models - wish vs. reality. In *High Performance Computing Systems and Applications, 2005. HPCS 2005. 19th International Symposium on*, pages 131–139, May 2005.

[18] A.C. Klaiber and H.M. Levy. A comparison of message passing and shared memory architectures for data parallel programs. In *Computer Architecture, 1994., Proceedings the 21st Annual International Symposium on*, pages 94–105, Apr 1994.

[19] Blaise Barney. Message passing interface (mpi). `https://computing.llnl.gov/tutorials/mpi/`. [Online; accessed 20140226].

[20] MPI: A Message-Passing Interface Standard. Version 3.0, 2012.

[21] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[22] R.L. Graham, G.M. Shipman, B.W. Barrett, R.H. Castain, G. Bosilca, and A. Lumsdaine. Open mpi: A high-performance, heterogeneous mpi. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–9, Sept 2006.

[23] R. H. Castain, T. S. Woodall, D. J. Daniel, J. M. Squyres, B. Barrett, and G .E. Fagg. The open runtime environment (openrte): A transparent multi-cluster environment for high-performance computing. In *Proceedings, 12th European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 2005.

[24] `http://www.mpich.org/`. [Online; accessed 20140319].

[25] Darius Buntinas, Guillaume Mercier, and William Gropp. Implementation and shared-memory evaluation of mpich2 over the nemesis communication subsystem. In Bernd Mohr, JesperLarsson Träff, Joachim Worringen, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4192 of *Lecture Notes in Computer Science*, pages 86–95. Springer Berlin Heidelberg, 2006.

[26] The mpich wiki. `http://wiki.mpich.org/mpich/index.php/CH3_And_Channels`. [Online; accessed 20140320].

[27] Mcapi A P I Specification, Document Id, Mcapi Api, Specification Document, and Release Distribution. Multicore Communications API (MCAPI) Specification, 2011.

[28] `https://bitbucket.org/hollisb/oopenmcapi/wiki/Home`. [Online; accessed 20140414].

[29] Shih-Hao Hung, Po-Hsun Chiu, Chia-Heng Tu, Wei-Ting Chou, and Wen-Long Yang. On the portability and performance of message-passing programs on embedded multicore platforms. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 896–903, May 2012.

[30] Linx protocols. `http://linx.sourceforge.net/linxdoc/doc/linxprotocols/book-linx-protocols-html/index.html`. [Online; accessed 20140225].

[31] `http://sourceforge.net/projects/linx/`. [Online; accessed 20140320].

[32] Patrik Strömblad. Enea multicore: High performance packet processing enabled with a hybrid smp/amp os technology. Technical report, 2009.

[33] Pierre-Louis Aublin, Sonia Ben Mokhtar, Gilles Muller, and Vivien Quéma. ZIMP: Efficient Intercore Communications on Manycore Machines. Technical report, INRIA Rhône-Alpes, 2011.

[34] Brian Paul Swenson and George F. Riley. A new approach to zero-copy message passing with reversible memory allocation in multi-core architectures. In *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, PADS '12, pages 44–52, Washington, DC, USA, 2012. IEEE Computer Society.

[35] Lei Chai, A. Hartono, and D.K. Panda. Designing high performance and scalable mpi intra-node communication support for clusters. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–10, Sept 2006.

[36] `https://lttng.org/`. [Online; accessed 20140226].

[37] `http://www.paratools.com/OTF`. [Online; accessed 20140206].

[38] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Introducing the open trace format (otf). In *Proceedings of the 6th International Conference on Computational Science - Volume Part II*, ICCS'06, pages 526–533, Berlin, Heidelberg, 2006. Springer-Verlag.

[39] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E. Nagel, and Felix Wolf. Open trace format 2: The next generation of scalable trace formats and support libraries. In *PARCO*, pages 481–490, 2011.

[40] `http://www.vi-hps.org/projects/score-p/`. [Online; accessed 201402017].

[41] `http://www.scalasca.org/`. [Online; accessed 201402017].

[42] `http://www.vampir.eu/`. [Online; accessed 201402017].

[43] Lttng userspace tracer. `http://lttng.org/ust`. [Online; accessed 20140602].

[44] S. Brehmer, M. Levy, and B. Moyer. Using mcapi to lighten an mpi load. *EDN*, 56(22):45–49, 2011.

[45] `http://sourceforge.net/projects/linx/files/MCAPI%20for%20LINX%20compatibility%20layer/`. [Online; accessed 20140414].

[46] `http://www.boost.org/doc/libs/1_55_0/doc/html/interprocess.html`. [Online; accessed 20140414].

[47] Boost c++ libraries. `http://www.boost.org/`. [Online; accessed 20140603].

[48] Boost interprocess: Sharing memory between processes. `http://www.boost.org/doc/libs/1_53_0/doc/html/interprocess/sharedmemorybetweenprocesses.html`. [Online; accessed 20140603].

[49] Yu-Hsien Lin, Chiaheng Tu, Chi-Sheng Shih, and Shih-Hao Hung. Zero-buffer inter-core process communication protocol for heterogeneous multi-core platforms. In *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA '09. 15th IEEE International Conference on*, pages 69–78, Aug 2009.

[50] Open cv. `http://opencv.org/about.html`. [Online; accessed 20140530].