Aspects of the efficient Implementation of the Message Passing Interface (MPI)

Jesper Larsson Träff

NEC Laboratories Europe, NEC Europe Ltd.
Rathausallee 10
D-53757 Sankt Augustin, Germany
traff@it.neclab.eu

Doctoral Dissertation
(Doktordisputats)

submitted for defense to the

Faculty of Natural Sciences
Institute of Computer Science, University of Copenhagen (DIKU)
Denmark

1. November 2007
Accepted for Defense 29. January 2009
Defended 2. October 2009

This doctoral dissertation ("doktordisputats") was submitted to the University of Copenhagen in November 2007. It was accepted for defense on January 29th, 2009. The official reviewers were Professor Jaswinder Pal Singh, Princeton University, USA, Professor Brian Vinter, Københavns Universitet, Danmark, and in part, Professor Peter Sanders, Universität Karlsruhe, Deutschland. All are thanked.

Minor typos and bibliographic errors have been corrected, a few stylistic improvements and technical clarifications undertaken, remarks to relevant developments regarding the MPI standardization process incorporated, and pointers to journal publications superseding four of the conference contributions of the dissertation inserted; otherwise the dissertation is printed as submitted.

Sankt Augustin, Germany, in April 2009

Jesper Larsson Träff

Denne afhandling er 29. Januar 2009 antaget til forsvar for den naturvidenskabelige doktorgrad ved Københavns Universitet.

Foreword

The following 26 technical research and overview papers, published in relevant scientific conferences and journals over a period of approximately 8 years, have been collected and commented toward the fulfillment of the requirements for the highest Danish academic degree of *Doctor Scientiarum* (Dr. Scient.). Together they constitute the so-called "Doktordisputats" (Doctoral Dissertation), hereby submitted for defense at the University of Copenhagen.

The work described in the dissertation has been carried out as the main activity of the author working since 1998 in the MPI team of the NEC Laboratories Europe (formerly C&C Research Laboratories), NEC Europe Ltd., in Sankt Augustin, Germany. The author wants to thank NEC for most generous working conditions, among those the possibility to work, although under difficult conditions, on the machine that for two years held the No. 1 position on the Top 500 list of the "500 most powerful computer systems" (see `www.top500.org`), the Japanese Earth Simulator that was inaugurated in 2002 and for which the C&C Research Laboratories developed the MPI library, and for the *freedom to publish* in outline the basic ideas and outcomes of this work. The opportunity to publish has provided the indispensable opportunities for interaction with the academic and technical community in high-performance, parallel computing, in which the message passing model continues to play a dominant role. The 1st Computer Software Division of NEC Japan is thanked for financial and technical support throughout the years. The author very concretely and warmly thanks the MPI team at the NEC Laboratories Europe, namely Hubert Ritzdorf, Andreas Ripke, Christian Siebert, Faisal Ghias Mir, and former team members Joachim Worringen, Maciej Gołebiewski and Rolf Hempel for inspiration, guidance and sometimes heated discussions on all matters MPI, as well as all other co-authors of the papers collected here. Thanks to Inge for love, patience, encouragement, support, indulgence and circumstantial despair over the years.

Sankt Augustin, Bonn, Yokohama, 1999–2007

Jesper Larsson Träff

This work is, finally, dedicated to the late Frank Zappa (1940–1993), Sun Ra (1914–1993), and Philip K. Dick (1928–1982)

"I hereby pronounce myself ready to go back on the road", Frank Zappa, Easter Sunday 1975, "One size fits all".

Contents

Overview

This dissertation is divided into three main parts. Part I puts message passing parallel programming into context, and provides a brief overview of the *Message Passing Interface* (MPI) standard, emphasizing the elements of the standard that have been in focus in the work presented as the technical contributions of the dissertation. Part II summarizes and discusses the individual contributions, and in a few cases provides additional and/or supplementary material. Finally, Part III contains (pointers to) the 26 technical contributions (*as they have appeared*) in the respective conferences and journals.

Part I

Message Passing with MPI

Chapter 1

Introduction

Message passing is a way of structuring and reasoning about parallel processes and programs, which is well suited to the way actual parallel machines have been and still are constructed. The message passing abstraction consists of a (finite) set of independent processes with local state that can exchange information only by explicit communication facilitated by a communication medium. A collection of processors with local memory and no common clock connected through a communication network and routing system which permits any two processors to communicate fits this description. This model and type of parallel computer is an almost exact opposite of models where a set of synchronized processors communicate directly via a common, shared memory.

The message passing model, which crystallized in the 1970ties and was practically developed throughout the 1980ties, has both theoretical and practical roots, theoretically in formalisms like CSP (*Communicating Sequential Processes* [Hoa78, Hoa85]), and various process calculi like CCS (*Calculus of Communicating Systems* [Mil80, Mil85]), practically in a number of language extensions and libraries for programming some of the parallel machines that were developed and marketed in the 1980ties: Intel Hypercube, IBM SP, Meiko Computing Surface, Thinking Machine Connection Machine CM-5, various Transputer based systems [Hoa91], and many others.

Despite many other, possibly more convenient parallel processing abstractions, especially for the design and analysis of *parallel algorithms*, typically presupposing a shared memory of certain power, with the *Parallel Random Access Machine* (PRAM) [FW78, JáJ92] being the most prominent representative, the message passing abstraction has prevailed as a model and practical framework for performance efficient programming of actually existing parallel machines. The *Message Passing Interface* [SOHL$^+$98, GHLL$^+$98], MPI for short, is a particular, practical incarnation of the message passing abstraction, which since its definition in the early to mid-nineties has assumed the status of *de facto* standard for message passing programming, especially in the area of "high-performance computing": scientific applications on dedicated, distributed memory systems with a significant number of processing nodes.

The work collected in this dissertation deals concretely with various aspects of the efficient implementation of the *Message Passing Interface*, mostly for a specific, hybrid, distributed-shared memory architecture. Although in this sense specialized, the specific algorithms

and the lessons learned are believed to be of more general interest and relevance, and the following overview chapters aim to show this. This summary is neither an overview of or a tutorial on message passing or MPI, nor a textbook on design and implementation of parallel communication algorithms, but has the more modest aim of putting a certain body of specific work into a somewhat broader context.

1.1 Background on parallel programming and performance models

Standard background in parallel processing and architectures is presupposed [GKKG03, PH04]. This section summarizes parallel machine and model classifications, and performance and design models for message passing systems as used in the following, both Part II and the technical papers collected in Part III.

Programming model: Depending on context, p usually denotes the number of processors, that are numbered (or *ranked*) from 0 to $p-1$. Each processor runs its own private *program*. The processors may run different programs, or it may be required that they all run the *same* program. The processors are not assumed to be synchronized, and therefore no assumptions about them executing "the same instruction at the same time" can be made. The processors operate on data stored in *local memory*. Following what could be called the *extended Flynn taxonomy* either an MPMD/MIMD (*Multiple Program Multiple Data/Multiple Instruction Multiple Data*) or an SPMD (*Same Program Multiple Data*) programming model is assumed. The Flynn taxonomy originated in [Fly72], and the SPMD assumptions were introduced in [DGNP88] (see also [Dar01]). By the SPMD assumptions data and code objects (like for example procedures) existing on one processor can also be referenced on any other processor, which is normally a prerequisite for programming models with *remote procedure calls*, for *Active Messages* [vECGS92], sometimes for the BSP model (see below), as well as for *Partitioned Global Address Space* (PGAS) languages. Anticipating the next chapter, MPI does not rely on this assumption, and it strictly supports an unrestricted MPMD/MIMD programming model.

Message passing abstraction: The *message passing abstraction* assumes as above a (finite), independent (non-synchronized) set of processors, each with only local memory, each executing a locally stored program, and that can communicate with each other by sending and receiving messages through a communication medium. The communication medium facilitates *reliable* communication between *any* pair of processors. This abstraction is supported by the underlying *communication network*.

Data parallel paradigm: The *data parallel paradigm* assumes that data are distributed in some pattern across a finite set of processors, each of which in a similar fashion process their individual part of the data. The processors may be tightly or loosely synchronized.

Network model: For the design and analysis of communication algorithms it is convenient to assume that the underlying *communication network* supporting the message passing abstraction is *fully connected* (that is, any two processes can communicate directly) and *homogeneous*, meaning that the cost of communication between any two processors is the same for all processor pairs.

Physically, this is true only for some (expensive) networks (like for instance full crossbars), and must otherwise be supported by routing in the underlying physical *communication network* [DYN03, DT04]. Many communication networks used in practice are physically non-homogeneous, e.g. meshes, tori, trees.

Important properties of a communication network is the communication *latency* or *start-up*, the *communication bandwidth* of a single communication link, the total or *bisection bandwidth* (defined as the minimum of the total bandwidth over all possible partitions of the processors into two roughly equal sized subsets when all processors of the two subsets are communicating), the diameter of the network, and many others which will be hidden under simplifying assumptions in the following.

One exception for which non-homogeneous communication structure will not be ignored are simple, hierarchical communications systems as found in SMP systems that have come into widespread usage since the late 1990ties. An SMP system consists of a number of nodes N, each with a (maximum) number of processors n, such that the total number of processors is $p = nN$. The nodes are interconnected by a physical communication network, and the processors on a node are connected to a shared memory that can be used for intra-node communication. It is assumed that communication between processors on a node is fully connected, has full bisection bandwidth (meaning here that all pairs of processors can communicate simultaneously at their full bandwidth regardless of how the processors are paired), and is homogeneous. Actual memory systems for SMP nodes with many processors are rarely that powerful. Communication between processors on the same node is typically (much) less costly than communication between processors on different nodes.

Communication capabilities: Each processor is connected to the network through one or more *communication ports* through which data are sent and received. If there is only one port communication is called *single-ported*, if there are k ports (that can be used simultaneously), obviously, *k-ported*.

The capability of each port, and of the network can be so as to allow communication in one direction only at a time. This is called *half-duplex*. If both sending and receiving is possible at the same time communication capabilities are *bidirectional* or *full-duplex*. Bidirectional communication may be restricted such that a processor can send and receive only from one other processor at a time, which is called the *telephone model* of communication. The most general model of bidirectional communication is to allow simultaneous sending to some processor and receiving from another, possibly different, processor. This model is sometimes referred to as the *(simultaneous) send-receive model* [BNK94a, BNKS00], and is supported by many current interconnects.

In SMP systems, the processors on an SMP node (typically) share the communication

capabilities, and it is the processor nodes that have the capabilities described above. These capabilities, as well as the communication bandwidth of the network are shared among the processors, which can obviously be a severe limitation for SMP systems with large nodes. For a formal model of (heterogeneous) SMP clusters, see for instance [CFMR05].

Complexity model: For analyzing the complexity of algorithms, in particular of collective communication operations in which all processors take part, it is a convenient abstraction to assume that communication takes place in synchronized *rounds*. The time for such a round is then bounded by the communication time of the slowest processor. This is an abstraction to ease the analysis, rarely actual (explicit) synchronization has to take place.

Performance model: There is a number of commonly used proposals for modeling the time spent in communication between two processors in a distributed system.

The simplest and most straightforward abstraction is the *linear cost* model. The time for transferring a message of size m (bytes or other unit) through the communication network is linear in m and modeled as $\alpha + \beta m$, where α is the *start-up latency* and β the transfer time per unit[1]. Because many communication systems, including MPI, for performance reasons use different protocols for different data sizes, the simple linear model is often not warranted. A *piecewise linear* model, with different α and β values for disjoint intervals of m, can sometimes be used for greater accuracy. Another problem with the linear model is that it models only transfer time and assumes that both sender and receiver are active for the whole duration of the transfer. This does not help in designing and analyzing algorithms with overlap between communication and other activities.

These deficiencies are addressed in the arguably more accurate (parametrized) *LogP* model [CKP+96], and its extension for long messages *LogGP* [AISS97]. The parameters in this model are *latency* L, *overhead* o in which a processor is involved in either sending or receiving a message, *gap* g which is the minimum time between consecutive message transfers, and the number of processors P. Because the *LogP* model was originally intended for small messages, parameters o and g were constants. For use with longer messages a *gap per byte* G was introduced in the extended *LogGP* model. These models are both subsumed in the *parametrized LogP* model of [KBV00]. In this model the time to send a message of size m is $g(m)$, and the message has been completed at the receiver at time $L + g(m)$ assuming that sender and receiver starts operation simultaneously, and that $g(m) > o(m)$.

Methods and benchmarks for measuring the *LogP* parameters for MPI implementations that are necessary for tuning of algorithms developed under this model were given in [BBC+03, KBV00, HMLR07].

A different way of modeling communication delays is the *postal model* [BNK94b, BNBH+95, BCD+96, BNK97], in which a message sent in round k arrives at its destination λ rounds

[1]For mysterious reasons the linear model is in the MPI community often called the "Hockney model" [Hoc94]. The model has been around for much longer as a first approximation to network performance.

later, in round $k + \lambda$. This model has sometimes been used for the design and analysis of collective communication algorithms.

Combined design, complexity and performance models: The *bulk synchronous processing* model [Val88, Val89, Val90b, Val90a] (BSP) combines aspects of design, complexity and performance models, and was intended as a *bridging model* between physically existing, practical parallel computers and more theoretical design and complexity models like the *Parallel Random Access Machine* (PRAM).

In the BSP model a computation is organized into a sequence of *supersteps*. In a computational superstep the processors perform computations on local data independently of each other. A computational superstep is followed by a communication superstep, in which data are exchanged among the processors, (logically) followed by a barrier synchronization. In a communication superstep each processor is assumed to receive at most h units of data and send at most h units of data, and the problem of scheduling and routing this communication is called an h-relation. The complexity of a BSP computation is the sum of the times for the supersteps. To model cost, the BSP model assumes two parameters, L which is the minimum time between two supersteps, and g (not to be confused with the g of the $LogP$ model) which is the computation to communication ratio, in addition to the problem size and the number of processors.

The *coarse-grained multicomputer* (CGM) [DFC+02] is a simplified BSP model. Another, recent model in this vein is the *parallel resource optimal* model (PRO) [GEL+06], which furthermore requires optimality relative to a sequential algorithm.

A number of libraries for BSP programming have been proposed and implemented [JW96, HMS+98, GLR+99, BJvOR03, Bis04], but they have not found widespread usage. The book [Bis04] discusses BSP application programming with BSPlib, and contrasts this to a BSP style implemented directly in MPI.

The paper [BHP+99] explores the relative (predictive) power of the BSP and $LogP$ models, and shows that they are substantially equivalent in that a $LogP$ computation can be simulated in the BSP model with constant slowdown, and a BSP computation can be simulated in the $LogP$ model with at most logarithmic slowdown (and constant for a wide range of model parameters).

The PRAM [FW78, JáJ92] is a different, abstract model for algorithm design and parallel complexity analysis. It assumes an unbounded set of processors operating in lock step with unit time access to a common, shared memory. The model has been theoretically enormously influential for investigating the inherent complexity of various problems and for the design of massively parallel algorithms [Rei93]. Many of the results, techniques and paradigms are relevant to algorithm design in other, weaker models, or for understanding inherent potentials and limitations of computational problems. Despite attempts of constructing various approximations and emulations of the PRAM [KKT01], the model in itself has been practically largely irrelevant.

Chapter 2

Towards a Message Passing Interface

The mid-80ties of the last century experienced a first upsurge in and enthusiasm about parallel computing, which was due both to enormous advances in microprocessor technology, and to certain climatic conditions of the time like the demand for performance implied by the socalled "Star Wars" program in the US, as well as by large scale scientific and environmental socalled "Grand Challenge" problems[1]. At the time a number of architecturally different systems were built, with both variants of shared memory (Denelcor HEP, 1982, Encore Multimax, 1985, Sequent Symmetry, 1987, Thinking Machines CM-2, 1987, MasPar MP-1 and MP-2, 1990 and 1992, Kendall Square Research KSR, 1991, and others) and distributed memory (Cray T3D, 1993, IBM SP, INMOS Transputer, 1984, Intel iPSC2 and iPSC/860, 1987 and 1990, Meiko CS and CS-2, 1986 and 1993, nCube, 1985, 1989 and 1995, Thinking Machines CM-5, 1991, and many others) [McB94]. Performance and scaling difficulties with shared memory systems gradually shifted the emphasis towards distributed memory systems with some form of message passing as the programming paradigm. This led to a growing consensus about a the need for a *portable* message passing interface.

Influential in the process were both large and small hardware and system vendors like Intel (with the iPSC hypercube), IBM (with the SP system), Meiko, nCUBE, and universities and research institutions like Argonne National Labs, University of Tennessee, GMD ("Gesellschaft für Mathematik und Datenverarbeitung" in Germany, no longer existing but taken over by the Fraunhofer Gesellschaft), and many others. Some of the individuals involved eventually got together in 1992 to define a message passing interface based on common ideas from some of the more successful interfaces in use at the time, and also with the aim of ensuring that developments in the US and in Europe would not diverge. This gathering developed into the MPI Forum that eventually took responsibility for defining the MPI standard. A first proposal for a *Message Passing Interface* was made available already in October 1992, and the first, final version of the MPI-1 standard was officially released in May 1994. Corrections to this standard were published in June 1995, at which point also work towards MPI-2 was started, resulting in the MPI-2 additions to MPI-1 being published

[1] A current, second and possibly much more desperate upsurge in parallel computing is driven by a relative decline in single-processor performance and the resulting transition towards multi-core processors.

in July 1997. The process leading to the MPI-1 and MPI-2 standard documents, including voting procedures, is vividly and carefully described by two of the founding members of the MPI Forum [Wal94, DOSW96, HW99].

MPI-1 defined a pure, although broadly scoped message passing model. It included point-to-point communication, communicators and process groups needed for building safe parallel libraries, collective communication, datatypes for describing complex layouts of data in memory, and the not strictly message passing process topology feature. MPI-1 defined bindings for C and Fortran 77, with bindings for C++ and Fortran 90 added with the first set of corrections in 1995. With MPI-2 additions that are not strictly message passing, but which fit into and significantly extend the message passing framework of MPI-1, were added. These include one-sided communication, parallel I/O, and dynamic process management.

Some influential precursors to MPI in use in the late 1980ties and early 1990ties, and some of the key concepts that they contributed to the standard should be mentioned. Some of these were non-portable vendor interfaces, others open, portable interfaces intended for a wider range of systems.

- PVM (*Parallel Virtual Machine*) [Sun90, SGDM94, GBD+94] is a message passing interface for programming in dynamic, heterogeneous environments, that was quite widely used in the early 1990ties. The dynamic process management features of PVM influenced MPI-2.

- The P4 macros [BBG+87, BL94] is a low level interface for both shared and distributed memory programming with a message passing abstraction. Since efficiency was a major goal of P4 it was later used as a low-level implementation interface for MPI.

- The portable, parallel macros PARMACS [CHHW94], partly evolved from P4, was a message passing interface which included a mechanism for defining virtual process topologies. The mechanism was adopted almost unchanged by MPI-1[2].

- The Zipcode interface [SSD+94] emphasized support for building safe, interference free parallel libraries. It introduced a communicator concept much like in MPI, and developed implementation mechanisms which were subsequently used in MPI implementations. Zipcode also had a virtual topology concept.

- The Express system [FK94] also stressed the importance of building high-level parallel libraries, completely hiding concrete message passing from the user. It had communicator and virtual topology concepts.

- Although not a message passing system Linda [CG89, CGMS94] is sometimes cited as an influence in the definition of MPI, maybe because its fundamental abstraction of an associatively addressed tuple space was so different.

[2]Apparently some versions of PARMACS allowed weighted edges in the graph topology (Hubert Ritzdorf, personal communication), but this unfortunately did not become part of the MPI specification.

- The XDR (external data representation) datatypes, also used in PVM for heterogeneous communication, influenced the definition of derived datatypes in MPI, and were also used in early implementations.

- IBM's *External User Interface* (EUI) [BBB+94] and *Collective Communications Library* (CCL) [BBC+95], were full-fledged message passing interfaces which substantially influenced MPI by their communicator concept (called *task groups*) and rich set of collective operations. Also the EUI introduced non-blocking send and receive operations with request handles and separate completion calls.

- Intel's NX was mainly a point-to-point communication library. It had tagged messages and selective receive operations. The library in its first version had no concept of process groups and collective communication.

- While all of the above mentioned systems were libraries or macros for existing languages, OCCAM [INM88] was a genuine message passing *language* (for the INMOS Transputer), based strictly on the CSP formalism. Message passing communication was synchronous with a construct for non-deterministic choice. The language had no concept of process groups, no collective operations, and little support for parallel libraries.

These and other approaches to message passing that were current at the time and all played a role in the process that led to MPI are described in a special issue of *Parallel Computing* which appeared in 1994 [HHMW94].

The goals of the MPI Forum was to develop an interface close to the then current practice to make a take-up by the various groups, vendors, and users likely. The interface should be *portable* over a wide range of platforms, but also efficiently implementable and fair in not favoring any particular machines' capabilities. Other main design goals were to support library building, and to allow the interface to be implementable in heterogeneous environments. The outcome was MPI-1.

MPI-1 met the design goals by centering on a small number of key concepts. Most important is the idea of safe communication domains in the form of the communicator construct, supported by process groups with local operations and collective operations for managing communicators. Another key concept was the flexible and compact derived datatype construct for representing application data structures. A point-to-point communication model whose "normal" send operation would not require intermediate buffering, and a comprehensive set of collective operations were the methods for communication. Despite the small number of key concepts MPI-1 ended up with a large number of functions, partly mandated by concerns for library building (attribute caching mechanisms), partly because the standard was also a compromise between many existing approaches with sometimes conflicting conceptions. The extensions introduced later with MPI-2 could build on the key concepts of MPI-1. The one-sided extension was a new communication model, the parallel I/O model could be seen as an extension towards external communication, and builds fundamentally on the derived datatype concept, while the dynamic process management facilities took up

on ideas that had been found valuable in for instance PVM. Interestingly, the MPI-2 process required more meetings than the MPI-1 process, sixteen instead of only seven.

2.1 Sociology and dynamics of MPI

MPI gained acceptance from its intended community quickly, and has over the past 15 years successfully developed into the *de facto* standard for message passing programming. It is interesting and worth pointing out that MPI is *not a standard* in the sense of being approved and administered by a standardization body like IEEE, ANSI or ISO. Instead, the standard is maintained by the MPI Forum, in which a number of (about 20) institutions has a vote for deciding about ballots of suggested corrections and minor improvements. The standard is the documents that can be found at `www.mpi-forum.org`. This consists of the MPI-1 documents, the corrections from 1995 called MPI-1.2, the MPI-2 document, and a list of accepted corrections to mostly parts of MPI-2. The first tree documents are more or less reprinted in two books [SOHL+98, GHLL+98], but there are some changes and additional explanations, and these books are *not* the MPI standard.

Currently the MPI Forum is in a bit of a stalemate situation. There are a number of open discussion points at the MPI mailing lists (which can be found mirrored at `www.mpi-forum.org`), many of which can be viewed as simple and relative uncontroversial corrections, and a list of Forum members entitled to vote on these, but votes are not cast. The list of MPI Forum members dates back to the MPI-2 meetings, following the then established procedures for entitlement (see [HW99]), and does not reflect the current forces in community. Furthermore, there is from various sides pressure for reviving the MPI process, although it is on the other hand not entirely clear what the issues of discussion should be. There is at the same time a genuine concern that the MPI standard may be weakened by accepting proposals and extensions in directions beyond efficient, dedicated message passing, that furthermore may not fit together.

Parallel with the definition of the MPI-1 standard, a first, paradigmatic implementation was developed, the so called MPICH [GLDS96] implementation from Argonne National Laboratories. It is generally recognized that this played a major and perhaps decisive role in making the standard a success. The prototype implementation could provide immediate feedback to the standardization process, and subsequently the MPICH implementation was used as a starting point for several vendor implementations, including NEC's MPI/SX. This meant that a reasonably well performing implementation of MPI-1 was available for use more or less in parallel with the standard itself. That this was not done for the MPI-2 extensions is probably one of the reason that the acceptance of this part of the standard has met with much more reluctance from both users and vendors. While complete implementations of the MPI-1 standard were available from many groups and vendors during the mid to late 1990ties, still only few really complete MPI-2 implementations are available, the first being announced in 2000, three years after the MPI-2 standard was made public.

During the standardization process ideas that were discussed and formalized to the point where they could have been included in the standard but voted out have been collected in

the socalled *Journal of Development*.

Aspects of message passing models and programming, the MPI standard itself, and, most significantly, the efficient implementation of aspects of MPI on classes of systems and specific machines has since many years occupied a central position in main stream parallel processing, and these themes are represented at all influential practical (and even theoretical) parallel processing conferences, e.g. ACM/IEEE Supercomputing, IEEE IPDPS (*International Parallel and Distributed Processing Symposium*), ICPP (*International Conference on Parallel Processing*), ACM PPoPP (*Principles and Practice of Parallel Programming*), Euro-Par (*European conference on Parallel Computing*), and occasionally even ACM SPAA (*Symposium on Parallel Algorithms and Architectures*). Since 1994 all aspects concerning the use, development and implementation of MPI (and PVM) have been the sole topic for EuroPVM/MPI, an annual conference dedicated to these message passing models. EuroPVM/MPI started out as the European Users Group Meeting on PVM, and shortly took up on MPI, which nowadays almost entirely dominates the conference.

The literature on all these aspects of MPI is enormous. There is no pretension of a systematic or exhaustive covering here. Pointers to literature are given where relevant for the discussion, where possible to more complete, extensive journal versions, that is with little intention of establishing precedence, and with indications of new or alternative developments of the discussed ideas. Care, especially in the papers in Part III, has been taken to give credit where credit is due.

2.2 Beyond MPI

MPI and the message passing abstraction are often criticized as too low level for productive (manageable, correct, scalable) development of parallel applications. There has been a large number of other proposals, still intended for actual, distributed memory systems with a communication network, that in various ways try to support higher level parallel programming abstractions. Some of these have been partly successful, mostly in smaller communities or for special, limited classes of applications. An early example is the above mentioned Linda abstraction [CG89, CGMS94]. For efficiency reasons, or for lack of generality, or both, none of these proposals so far have been able to supersede MPI as a common basis for portable, parallel programming of distributed memory systems.

Another early attempt at a more convenient programming abstraction was the data parallel High Performance Fortran (HPF) [Hig93, Sch97] definition. In the data parallel paradigm the distribution of data over the individual processors implicitly determines exchange of data which thus need not burden the application programmer, and High Performance Fortran would ideally allow programming at a higher level of abstraction (various kinds of parallel loops). Another idea was not to make changes to the host Fortran language but to express the division of data and computations by pragmas to be exploited by the compiler. Thus, a High Performance Fortran program would also be executable as a sequential program on a single processor. This presumably worked reasonably well for limited, highly structured applications with regular data distributions, but failed for more irregular computations. Since

the beginning of the century the influence of High Performance Fortran has been very limited. It is worth mentioning that the organization of the HPF Forum did inspire the rules of the MPI process: formation of subcommittees, regular meetings, rules of voting by commitment.

More general mostly data parallel language additions by means of pragmas and some library support are found in OpenMP [DM98, CDK+01] for shared memory systems. OpenMP has been more successful than High Performance Fortran because of the shared memory restriction, but do run into some of the same problems for irregular computations. Efficiency suffers, and the initially elegant idea of annotations to the normal language constructs by means of simple and clean pragmas begin to look very untenable. Pragmas becomes highly complex with obfuscated semantics, and examples with more pragmas than program lines are common. There has been extension of OpenMP (implementations) to distributed shared memory, but so far with unsatisfactory performance, and such extensions are not widely in use.

More recently, socalled *Partitioned Global Address Space* (PGAS) languages have gained attention. These are more flexible than the strict data parallel paradigm, and are at the same time claimed to be more efficient, partly because they are languages which can profit from many genuine advances in compiler technology. The best known examples are UPC [EGCSY05], Titanium [YSP+98, YHG+07], and Co-Array Fortran [RN07, CDMC+05]. An earlier, library based framework in the same vein was Global Arrays [NHL94, NH97].

Two other, but even more wide ranging languages are currently coming out of the US Government (DARPA) sponsored *High Productivity Computing Systems* (HPCS) initiative, which aims at both productivity and performance [LY07]. The two currently funded developments in the initiative are X10 [SSvP07] from IBM, and Chapel [CCZ07] from Cray. It is not to be expected that these languages will be in any way successful in the sense of being taken up by a large community of application developers, simply because they are not portable beyond their narrow range of target machines, and because by being (quite ambitious) languages the effort required to make them so is large. Their impact if any will at most be indirect.

All of the above mentioned languages and libraries assume an SPMD model.

Chapter 3

An Overview of MPI

In the following we give an overview of MPI as background for the technical contributions of Part III. This overview is not intended as a tutorial of MPI. To use but also to gain a deeper understanding of MPI the two books which essentially contain the standard document [SOHL+98, GHLL+98], with plenty of *Rationale*, *Advice to users* and *Advice to implementers* are the place to start. A number of tutorials [GLS94, GLT99] and several more general textbooks on parallel programming with message passing using MPI [Pac97, Qui03, KK03] (to mention but a few) are available. The ongoing discussion on corrections (and possible extensions) to the standard (both MPI-1 and MPI-2) can be found in the resources provided by the MPI Forum at

`www.mpi-forum.org`

From now on the term MPI means the full standard. Only rarely a distinction between MPI-1 and the MPI-2 extensions will be made. On September 4th 2008 the MPI Forum released the consolidated MPI 2.1 standard document [MPI08] which merges these two disparate parts of the standard into a (more) unified whole. This is the current version of the MPI standard, and all further versions (MPI 2.2 to be expected by the end of 2009, possibly later an MPI 3.0) of the MPI standard will build on this document.

MPI is a library with interfaces for the programming languages C, C++ and Fortran. This division of work means that no modifications to those languages or their compilers are necessary, and ensures a high level of portability. A program which has been implemented for one system with a standard conforming MPI implementation should be portable to any other system with MPI and requires recompilation at most. The drawback of a library approach is that some duplication of work between application and library may be introduced, and that local, dynamic state of the computation may have to be explicitly (and repeatedly) passed back and forth between application and library. This is sometimes tedious and unnatural, and is the case for some aspects of MPI, for instance with respect to structured data (see Section 3.2). Other drawbacks of the library approach compared to a parallel programming language are loss of efficiency and lack of elegance, as pointed out in the curious and one-sided critique in [Han98] from the standpoint of CSP-like, synchronous communication semantics. This critique severely underestimates the express need for portability and a more flexible

semantics in line with the approaches, languages, and libraries for message passing around at the time when MPI was defined, and that could most readily be met with the library approach taken by MPI.

MPI is primarily concerned with communication, which is expressed by explicit library calls. MPI is carefully designed to allow implementation of higher level, *safe*, parallel, application specific libraries. To this end some amount of additional "bookkeeping" functionality is built into the interface for handling and annotation of the various distributed objects defined by MPI. Mostly such functionality, e.g. attribute caching on objects, will not be further described here.

MPI is designed to support the most general MPMD/MIMD programming model in which the processors may run different programs. Typically, this can happen when additional MPI processes are spawned dynamically, although also static applications (in which the number of MPI processes is fixed from the outset) can make use of this freedom. However, many applications are programmed in the SPMD model, and each process executes the same program (of course without any implied synchrony). MPI is also, primarily by its datatype mechanism, designed to support heterogeneous systems where the representation of objects may be different among the processors of the system.

MPI is a process oriented interface (although thread based implementations have existed [Dem97, TSY00, TY01]). Processes are normally statically bound to a static set of processors, and communication between MPI processes takes place solely by the communication operations of MPI. Processors may run more than one MPI process.

Three major *communication models* or *paradigms* are defined by MPI.

- In the *point-to-point* model a *sending process* sends data to an explicitly specified *receiving process* which is also explicitly involved in the communication (Section 3.3).
- In the *one-sided* model an *origin* process initiates *get* or *put* (or *accumulate*) communication with a *target* process that is not explicitly involved in the communication (Section 3.4).
- In the *collective communication* model all processes take part in a joint data exchange or computation operation (Section 3.5).

MPI defines two additional models or paradigms, that are not strictly for communication.

- A *parallel I/O* model allows MPI processes to read and write data to and from external files, either individually or collectively (Section 3.7).
- A *dynamic process management* model makes it possible for MPI processes to dynamically create (spawn) new processes, and for independently started processes to establish MPI communication with other MPI processes (Section 3.8).

All communication in the MPI communication models is between memory segments or *buffers* that have been allocated by the application in the *user memory space* of the MPI

processes. Communication buffers are managed by the application. Communication buffers are untyped addresses passed to MPI in the communication calls, and stores either contiguous sequences of primitive data elements, corresponding to the primitive datatypes of C, C++ or Fortran, or more generally non-contiguous sequences of structured data. MPI provides a concise and very powerful mechanism to describe arbitrary layouts of data in communication buffers. This mechanism is called *derived* or *user-defined datatypes* (Section 3.2).

MPI realizes and supports the message passing abstraction of a fully connected, distributed system, in which all processes can communicate by the same mechanisms without exceptions. In order to be implementable on as wide a variety of actual, physical systems as possible, MPI does not incorporate a performance or cost model in any form, and does not provide any performance guarantees whatsoever. As witnessed by the large number of systems on which MPI has actually been implemented, from networks of workstations with very weak communication systems to dedicated, high-performance systems with custom communication networks, this has been a wise design decision. The only way to determine how an MPI implementation performs on a given system is by benchmarking. In Chapter 11 some possible drawbacks of the lack of performance model or guarantees are discussed.

By abstracting away from the underlying system the capabilities of the communication system are not in any way reflected in MPI, and it is not possible for an MPI application directly to control the allocation of processes to processors so as to improve communication performance. MPI makes one concession to this. By the means of virtual process topologies the communication pattern of (a part of) an application can be described in the form of an unweighted communication graph and passed to MPI which has the possibility of suggesting a reordering of the processes which may improve performance for the specified communication pattern.

The three communication models of MPI (as well as the parallel I/O) model specifies *progress rules* which state under what conditions communication must be guaranteed to progress and eventually terminate. These rules have been carefully formulated with a degree of interpretational freedom but to generally allow correct implementations of MPI without a dedicated progress thread (per process) and without special hardware support for handling communication (apart from the communication network, of course). Despite the defensive and careful formulation of the progress rules, the issue of progress is regularly a matter of controversy. MPI does not guarantee *fairness* in communication.

Although care was taken to specify MPI to allow implementations without intermediate buffering (in particular the "normal" send function that is not required to buffer, and therefore may lead to deadlock if crossed by another send from the destination process, but also the collectives were defined with that in mind), MPI is still a memory demanding interface. This is due to powerful constructs for representing sets of processes, and in particular to the fact that messages may arrive at a process *unexpectedly*, before a corresponding operation to receive the message was posted. For efficiency reasons this mandates some amount of hidden, intermediate, unbounded buffering.

Although MPI defines error classes and codes which can be returned to the user, it is quite explicit in stating that errors *may*, but *not* that they *will*, be handled by an implemen-

tation. On the other hand the standard is (mostly) very precise in stating for each of the MPI functions the conditions for a call to be correct. Error handling was left optional for efficiency reasons. When/if an error is detected, an error handler is called, and MPI provides mechanisms for the user to define handlers.

A major design goal of MPI was to allow the implementation of *safe* parallel libraries. By this is meant that it must be possible to guarantee that communication relating to the library does not interfere with communication of the application performed outside of the library. Furthermore it must be possible for the library to query and decode MPI objects created by the application that are passed to the library. The basic construct for separating communication domains is the communicator abstraction, but also a very elaborate attribute caching mechanism was introduced for this purpose. Attributes can be attached to all the main types of (distributed) objects like communicators, windows, file handles, and datatypes. It is also possible to decode derived datatypes to determine the sequence of constructors used to create the type.

MPI is, in principle but not formally so *self-simulating* [KRS90]. This means that it is possible to implement MPI in MPI, but only if a lot of coding of MPI objects is done manually. The self-simulation property is relevant, because MPI is intended for library building, and also because MPI is sometimes used to implement new collective communication algorithms, or to test improved implementations of collectives already defined in MPI, or to repair deficient implementations of MPI collectives. Some of the main impediments to self-simulation, that are also of practical concern, are the following.

- derived datatypes are not first-class citizens, that is they cannot be sent and received in communication operations, and they were, until the introduction of decoding functionality with MPI-2, completely opaque.
- the address type `MPI_Aint` is not a first-class citizen, that is there is no MPI type (like `MPI_INT`) describing `MPI_Aint`, so objects of type `MPI_Aint` cannot formally be used in communication and other operations.
- The operator `MPI_Op` and other similar objects are not first class citizens.
- The pack and unpack functionality (MPI_Pack and MPI_Unpack) is restricted and does not allow arbitrary breaking of buffers of structured data into blocks. This makes the implementation of blocked and pipelined algorithms working on structured data difficult for the user.

Finally, MPI is designed to allow coexistence with other parallel programming paradigms, and is used routinely on SMP systems together with OpenMP [CDK+01] and other shared memory paradigms. It is also designed to coexist with *threads*, which is a more complicated issue. The standard allows for implementations of MPI that are *thread-safe* at various levels. An implementation may allow only a single thread (`MPI_THREAD_SINGLE`), may allow a master thread to call MPI functions (`MPI_THREAD_FUNNELED`), may allow an arbitrary thread to perform MPI calls but only as long as no other thread performs an MPI

call at the same time (MPI_THREAD_SERIALIZED), or may support unrestricted thread safety (MPI_THREAD_MULTIPLE). A recent analysis of the MPI standard in [GT07a] elaborates on what is required and what is guaranteed from an MPI implementation to support threads.

3.1 Processes and communicators

A basic concept in MPI is a *distributed data structure* (or *distributed object*) for representing *ordered* sets of processes, called a *communicator*. Only processes belonging to the *same* communicator can *communicate*. A communicator actually consists of a unique *communication context* and an *ordered* set of processes called a *group*.

Communicators in MPI come in two flavors with different capabilities, namely *intra-communicators* and *inter-communicators*. Here, mainly *intra-communicators* are of concern (for inter-communicators, see Section 3.8). Distributed data structures (other examples are *windows*, see Section 3.4, and *file handles*, see Section 3.7) can only be manipulated by special, *collective* operations (see Section 3.5).

The communicator concept is one of the basic means of MPI for creating *safe*, interference free parallel libraries. To ensure safety, a parallel library duplicates the communicator used in the initialization of the library and/or in individual library calls, and by using this new, distinct communicator, communication inside the library is guaranteed not to interfere with communication between the same processes taking place outside of the library (in the application or in other libraries).

The number of processes in a group/communicator is called the *size* of the group/communicator, and each process in the group/communicator is identified by a unique integer *rank*. Process ranks within a group/communicator are always consecutive, starting from 0.

When an application is started the MPI library must be initialized by calling the MPI_Init function, before which no MPI functions (with exception of the query function MPI_Initialized) can be called. Among many other things the MPI_Init call creates a communicator called MPI_COMM_WORLD containing all processes specified (externally) in the launch of the parallel application. From MPI_COMM_WORLD new (sub)communicators can be created by various collective operations. The process group associated with a communicator can be extracted locally by the MPI function MPI_Comm_group.

Process *groups* that represent ordered sets of processes are MPI objects that can be manipulated locally, that is independently by each MPI process, and for instance used in the (collective) creation of new communicators. Operations on groups include forming new groups by unions and intersections, finding a process' local rank in a group, and translating ranks among groups.

A process group represents a (usually static) *process to processor* mapping. To facilitate fast (constant time) translation from group local ranks to processor identities, this mapping is often implemented by an array of size p indexed by rank. This is expensive in memory, theoretically not scalable, and can be problematic in practice for systems with a large number of processors and comparatively scarce memory resources – as is the case for the IBM Blue Gene/L system [MSA+07, FCLA06] (Gheorge Almasi, personal communication).

3.2 Datatypes and memory layout

MPI allows communication not only of data which occupy consecutive locations in memory (simple buffers), but also of non-consecutive data scattered over a larger segment of the memory of a process. Non-consecutive data arise in applications by the use of the various structuring concepts offered by the implementation language of the application. Being a library, the application specific layout of data must be explicitly conveyed to the MPI library. This is done through the (recursive) use of constructors which make it possible to describe arbitrarily complex layouts of data. This mechanism of MPI is called *derived* or *user-defined datatypes*.

Communication (and file) buffers in MPI are described by three parameters:

1. the start address of the `buffer`,
2. a `count` of the number of consecutive elements in the buffer, and
3. a `datatype` describing the layout or structure of each element

Such an MPI communication buffer will be denoted by a triple `[buffer,count,datatype]`.

The element data layout is described by an MPI datatype which can be either a basic (simple) datatype or a derived type as mentioned above.

The basic datatypes correspond to the simple datatypes found in the host languages C, C++ or Fortran, e.g. `MPI_INT` for (C `int`) integer types, `MPI_FLOAT` and `MPI_DOUBLE` for single and double precision floating point numbers of different precision, `MPI_CHAR` and `MPI_BYTE` for byte sized types, etc. Each of these types occupies a small, consecutive unit of memory, e.g. a byte, a word, a double precision word, etc.

In general, a data layout is a sequence of basic datatypes, called a *type signature*, and a corresponding sequence of *displacements* or offsets, describing where each basic datatype is to be placed relative to the beginning of a buffer. The type signature together with the sequence of displacements is called a *type map*, thus the terms derived datatype, data layout and type map can be used more or less synonymously.

The *size* of a datatype is the total size of the basic datatypes in the type signature in some unit (typically bytes). The *extent* of a datatype is the difference between the smallest displacement and the largest displacement plus the size of the corresponding basic datatype in the type map. For an MPI type `mpitype`, extent(`mpitype`) will be used to denote the extent of the type, and is used for explanatory purposes in Section 3.5.

Derived types are recursively built up by the use of constructors, MPI functions that must be explicitly called, and generate type maps according to rules associated with each constructor. Let T_i be already defined datatypes and $c_i \geq 0$ *repetition counts* for $i = 0, \ldots$.

- MPI_Type_contiguous creates a *contiguous type* of c_0 instances of the component type T_0, each instance starting at the offset where the previous instance ended.

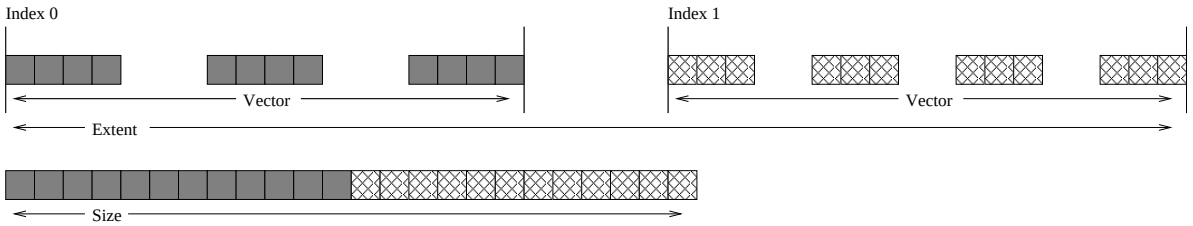

Figure 3.1: A data layout in memory and *type map* of a structured type built over two vector types. Also shown is the conceptually existing *type signature*.

- MPI_Type_vector, MPI_Type_create_hvector create a *vector type* consisting of c_1 strided blocks, each consisting of a contiguous sequence of c_0 instances of the component type T_0.
- MPI_Type_indexed, MPI_Type_create_hindexed, MPI_Type_create_indexed_block create an *indexed type* consisting of a number of contiguous blocks of c_i instances of the component type T_0, each block starting at a specified index.
- MPI_Type_create_struct creates a *structured type* consisting of a number of contiguous blocks of c_i instances of potentially different component types T_i, each block starting at a specified displacement.

MPI in addition has constructors for *distributed array types* and *subarrays*, but these can (easily) be defined in terms of the constructors explained above.

An example of a structure consisting of two vector types is shown in Figure 3.2.

Using the description of the type constructors, an explicit type map can easily be constructed, and some MPI implementations or parts thereof, like the ROMIO [TGL98, TGL99b, TGL99a, TGL02] implementation of parallel I/O, do represent datatypes by their type map. This is a space consuming representation proportional to the number of basic datatypes in the type map, which depends directly on the actual size of the repetition counts used to build the type.

A much more compact representation is as a DAG (*directed acyclic graph*) consisting of nodes for the constituent basic types and one (or two) node(s) for each constructor call used when creating the type with the corresponding repetition counts and displacements on the edges from constructor node to component type node. For example a contiguous type has one child, representing the component type, a vector child one child representing the contiguous block making up the vector (which again has one child), an indexed type multiple edges to the same component type, each with different repetition count and index, and, finally, a structured type several children for each component type.

This representation is clearly a DAG and not a tree since the same component type can occur many times at different positions. By a *leaf* of such a DAG is meant a node with no children. Leaves are clearly basic types. The *root* is the single node with no incoming edges.

The *number of leaves* L of such a DAG is the number of different traversals from root to a leaf. During the traversal to the leaves a tree T representing the DAG can be built.

The DAG representation leads naturally to a recursive formulation of the routines needed to traverse and process the basic datatypes of a derived datatype, for instance for packing and unpacking into and from consecutive buffers. In such a straightforward recursive formulation, the internal repetition counts can make the number of recursive calls very large, and in particular unbounded in the number of leaves L.

MPI provides functionality to pack and unpack structured data into or from consecutive buffers. The MPI_Pack and MPI_Unpack functions however only allow packing or unpacking into or from full MPI buffers `[buffer,count,datatype]`. There is no functionality to navigate inside a non-consecutive buffer, and no functionality to pack or unpack only parts of a structured buffer. This makes it difficult to implement advanced pipelined algorithms (operating on non-consecutive data) in MPI. To assist in library building where datatypes might be given as black box arguments, MPI provides a set of decoding functions, which can be used to (recursively) extract the component types of a derived datatype. This functionality is awkward and clumsy.

For the application programmer it can be a tedious exercise to describe once again (the layout of) the application datatypes to the MPI implementation, and a better integration of the MPI datatypes with the target languages would be desirable. Tools have been proposed and developed to aid with the setting up of the MPI datatype constructor calls directly from the application code [GMPD98, MSD99, HI00, RP06], but seem limited and sometimes make rather arbitrary assumptions. Such tools have not found widespread usage.

It is worth pointing out that the MPI datatype mechanism can only describe a known layout of basic datatypes in memory. To use the mechanism with high-level programming language, data-dependent data structures (lists, trees, ...) it is the responsibility of the application programmer to traverse such structures and determine the whereabouts in memory of the various components. MPI provides no help with that.

3.3 Point-to-point communication

The point-to-point communication model defines the operations for explicit communication between two MPI processes, one of which sends data and one of which receives data. For communication between two processes to take place the send and receive operations must match, which is captured in a point-to-point *matching rule*. The processes must belong to the same communicator. The communication is named in that the sending process specifies the *rank* of the receiving process, and the receiving process the *rank* of the corresponding sending process – or a wildcard allowing reception from an arbitrary process in the communicator (`MPI_ANY_SOURCE`). Furthermore, a message is sent and received with a tag, which must be identical for the communication to match – again a wildcard allows reception of a message with an arbitrary tag (`MPI_ANY_TAG`). Messages are *non-overtaking*. If two messages are sent with the same tag to another process, they are received in the order in which they were sent. Data to be sent and received are specified by triples `[sendbuf,sendcount,sendtype]`

and [recvbuf,recvcount,recvtype], respectively, as explained in Section 3.2. Finally, the type signature of the data sent must be a prefix of the signature of the data to be received. This means that the types of the basic elements sent must be the same as the initial sequence of types of basic elements specified by the receiver. Note, however, that it is not required that the number of basic types sent and received is the same. In MPI point-to-point communication it is allowed to send less data than expected by the receiver. The number of actually received basic and typed elements can be queried by the receiver with the MPI_Get_count and MPI_Get_elements functions.

MPI send and receive functions are either *blocking* or *non-blocking*. A call to a blocking send function returns when the data to be sent have been copied out of the send buffer (triple). There is no implication neither on whether the data have actually been sent by the communication network nor on whether the receiving process has started to process the data. A call to a non-blocking send operation only prepares the MPI library for the communication, and will return immediately. Likewise, a non-blocking receive only prepares the MPI library for the communication. In MPI terminology, non-blocking operations are therefore called *immediate*.

MPI specifies four different blocking *send modes* with differing semantics and corresponding send functions:

- MPI_Send(sendbuf,...,rank,tag,comm). The *normal* send initiates the sending operation and returns when the send buffer can be used again. There is no implication that a matching receive has been posted, and the normal send may or may not complete depending on the action at the receiving side.

- MPI_Ssend(sendbuf,...,rank,tag,comm). The *synchronous* send operation initiates a send operation, waits for the matching receive to start, and returns when the send buffer can be reused. This notion of a synchronous send is less strict than the notion found in for example CSP. In MPI the send call may return before the receiver has completed. The only semantic guarantee is the start of the receive operation.

- MPI_Bsend(sendbuf,...,rank,tag,comm). The *buffered* send operation must complete irrespective of the action at the receiving side. If necessary, data are copied into a temporary buffer, that must have been preallocated by the user. This type of send corresponds to the send operation of previous message passing libraries, for example Intels NX, and was for that reason included in MPI.

- MPI_Rsend(sendbuf,...,rank,tag,comm). The *ready* send operation initiates a send under the assumption that a matching receive has already been posted, and returns when the send buffer can be reused. This operation was introduced because the assumption of an already posted matching receive in some cases allow saving of acknowledgment messages. Also no unexpected messages will be generated.

All four modes can also be non-blocking, in which case the mode name is prefixed with an I: MPI_Isend, MPI_Issend, MPI_Ibsend and MPI_Irsend.

The blocking receive operation – there is only one! – looks similar:

- MPI_Recv(recvbuf,...,{rank|MPI_ANY_SOURCE},{tag|MPI_ANY_TAG},comm,status) receives matching data into the specified buffer, and returns with information on the *status* of the operation in the `status` argument when reception is complete.

The receive operation can also be non-blocking. After completion of the receive operation the number of elements actually received can be read from the `status` object (with either MPI_Get_count or MPI_Get_elements).

The non-blocking send and receive operations return immediately with a *request* object which can be used to query the state of the communication operation and eventually to enforce completion (in the same sense as for the corresponding blocking operations). For this a number of test (MPI_Test, ...) and completion (MPI_Wait, ...) operations are defined. Blocking and non-blocking send and receive operations can be mixed. A message sent by MPI_Issend can be received by MPI_Recv, and so on and so forth. MPI also provides facilities to probe for arrival of specific messages (MPI_Probe), setting a `status` argument which among other things contains the size of the arrived message. MPI also incorporates a (semantically hazy) possibility of canceling point-to-point communication operations.

Send and receive operations between two fixed processes is deterministic (messages are non-overtaking), but the wildcards `MPI_ANY_SOURCE` and `MPI_ANY_TAG` introduces non-determinism. The non-blocking calls also introduce non-determinism. For example several posted non-blocking receives from different processes may complete in any order, dependent on when the corresponding sends take place. To eliminate a nasty sort of indeterminate results, receive datatypes must be *non-overlapping*. This means that the elements of the type map must all have distinct displacements, and no displacement must be within the space taken by the basic element stored at some other displacement. Send types do not have this restriction.

Although MPI defined the normal send operation so as not to require buffering, the point-to-point communication model can still be expensive in terms of memory requirements. In most MPI implementations each process allocates a small, fixed number of slots for message headers and short messages (so for instance MPI/SX, see Chapter 6), which in principle does not scale with the number of processes. Since a message can arrive "unexpectedly" before a matching receive has been posted, space will be required to store at least information on the message, and this can grow unboundedly. In the paper [CK96] a protocol with only constant space and a constant amount of communication per message is developed for message passing allowing only *selective receive* (that is, no wildcards). It is then shown that to implement the MPI-like model with *non-selective receive* (that is, allowing `MPI_ANY_SOURCE`) takes either non-constant space or requires a non-constant amount of communication per message.

To ensure that communication eventually takes place, without committing to a specific implementation or posing unreasonable hardware requirements MPI formulates *progress rules* for point-to-point communication. In particular, MPI is (presumably) defined so as to allow implementations that do not have a separate progress engine taking care of communication in the background. For blocking communication the progress rule says that "if a pair of matching send and receives have been initiated on two processes, then at least one of these two operations will complete, independently of other actions in the system" [SOHL+98, Page

55]. For non-blocking communication the progress rule says that "a call to MPI_Wait that completes a receive will eventually return if a matching send has been started, unless the send is satisfied by another receive. In particular, if the matching send is non-blocking, then the receive completes *even if no complete-send call is made on the sender side*. Similarly, a call to MPI_Wait that completes a send eventually returns if a matching receive has been started, unless the receive is satisfied by another send, and *even if no complete-receive call is made on the receiving side*" [SOHL+98, Page 82] (authors' emphasis).

This seems to enforce some kind of independent progress mechanism, contradicting the intention of the standard to be implementable without any such. It would have been possible to formulate a completely sensible, slightly less demanding progress rule, but this is a contentious issue, often causing discussion, and is deliberately left somewhat vague in the MPI standard.

3.4 One-sided communication

The one-sided communication model makes it possible for processes to access regions of memory of other processes without the explicit involvement of the processes whose memory is being accessed. The memory regions which are being exposed to other processes for one-sided access are represented in MPI by a distributed object called a *window*. A window is created and destroyed by collective operations MPI_Win_create and MPI_Win_free. A window object includes for each process a start address in user memory of the process, the size of the memory region and a displacement unit for computing offsets.

In one-sided communication operations the process accessing the memory of another process is called the *origin* and is responsible for providing all arguments of the communication. The accessed process is called the *target*. Communication arguments consist in an MPI communication buffer of the origin process `[origin_buf,origin_count,origin_type]`, whereas the communication buffer at the target is described by an offset relative to the local address of the exposed memory region, `[target_offset,target_count,target_type]`.

One-sided communication can take place only within a socalled communication *epoch*. A process wanting to access memory of other processes must be in an *access epoch* and a process being accessed must expose its memory by being in an *exposure epoch*. Epochs are opened an closed by the one-sided *synchronization operations*.

The one-sided communication model provides three communication operations.

- MPI_Put transfers data from the user buffer at the origin to the buffer in the window at the target.
- MPI_Get transfers data from the user buffer in the window at the target to the user buffer at the origin.
- MPI_Accumulate performs a binary operation on data from origin and data from target with the result stored in the target buffer. The operation is atomic on the level of basic

datatypes. The operation can be any of the MPI predefined operators for reduction collectives (see Section 3.5).

It is not defined when an individual one-sided communication operation takes effect. When an access epoch is closed, all one-sided communication operations performed within the epoch must have been completed at the origin. When an exposure epoch is closed, all one-sided communication operations for which the process was a target must have been completed. The MPI standard gives the implementer a lot of freedom as to when communication is actually performed. In particular, it is possible to postpone and group several accesses to the same target within an epoch. The precise semantics are quite intricate in order to allow implementation of the model on different systems [GHLL+98, Page 136], and subject of some debate.

The *progress rule* for one-sided communication is as the rule for non-blocking point-to-point communication, in particular any communication that has been enabled must eventually be completed upon termination of the epoch.

To preserve some degree of determinacy it is in general not allowed to update the same memory location in a target window more than once within the same epoch. For derived datatypes this has the consequence that target types used in MPI_Put operations must have non-overlapping type maps.

The one-sided model defines three synchronization methods. Target processes can be either *active* or *passive*. In active synchronization both origin and target are involved in the opening and closing of the corresponding epochs. In passive synchronization (locks) the origin alone is responsible for all synchronization. The synchronization operations are summarized in Table 3.1.

For active targets two synchronization methods are defined. In *scalable* synchronization, termed so here because the method is collective and permits an efficient implementation, all processes in the communication window calls the MPI_Win_fence operation. The call closes a previous epoch and starts a new epoch that can be used both for access and exposure by all the processes. In the *dedicated* method each origin requests access to a set of targets specified in the call (MPI_Win_start), and each target grants exposure to a set of processes likewise specified in the call (MPI_Win_post).

The one-sided model is suited to implementation of BSP applications. For small h-relations the dedicated synchronization method should be used, for large or very irregular h-relations scalable synchronization is presumably preferable.

It should be noted that the MPI defined locks are *not* locks in the traditional sense of providing access to a critical section. The mechanism is too weak to readily permit atomic read and update of data on the target. This is an often criticized omission in the one-sided communication model. This is discussed in more detail in [GLT99]. Partly a consequence of this, partly a consequence of strong semantic restrictions on concurrent updates to windows (simultaneous local and remote updates are generally disallowed), the one-sided model of MPI is not suited as a compilation target for PGAS languages, as analyzed in [BD04].

It is possible to provide hints on the nature of communication and synchronization to the synchronization operations in the form of socalled *assertions*. These assertions allow for

Type	Name	Origin	Target
Active	Scalable	MPI_Win_fence	MPI_Win_fence
	Dedicated	MPI_Win_start MPI_Win_complete	MPI_Win_post MPI_Win_wait
Passive	"Lock"	MPI_Win_lock MPI_Win_unlock	

Table 3.1: The three one-sided synchronization operations.

optimizations in some cases. For instance assertions `MPI_NO_PRECEDE` and `MPI_NO_SUCCEED` state that no epoch precedes the epoch being opened or that no epoch succeeds the epoch being closed by a MPI_Win_fence call. The `MPI_NO_CHECK` assertion in MPI_Win_start and MPI_Win_post, and in MPI_Win_lock, likewise make it possible to save acknowledgment messages in these mechanisms. Likewise, on creating a window it is possible to give hints to the MPI implementation, for instance that no locking will be performed on a process. For some implementations this can save the spawning of a separate progress thread.

3.5 Collective communication

Collective operations in MPI are operations in which all processes in a communicator are involved, and jointly perform an operation. The *collective communication operations*, or *collectives* for short, are the 16 functions for *synchronization* (MPI_Barrier), *data exchange* (MPI_Gather, MPI_Scatter, MPI_Alltoall, ...), and *reduction* (MPI_Reduce, MPI_Allreduce, MPI_Scan), but also a number of bookkeeping operations for creating and destroying communicators, either explicitly or implicitly, as well as file and dynamic process management operations are semantically collective. A definition of the meaning of *collective operation* in MPI terms is given below. We let i and j denote process ranks in an *intra-communicator* `comm` of size p, so that $0 \leq i < p$ and $0 \leq j < p$.

Definition 1 (Collective MPI operation) *If process i in communicator `comm` is calling collective operation* MPI_*A, then all other processes in `comm` must eventually call* MPI_*A, and no other collective operations on `comm` may be called before* MPI_*A. Collective operations in MPI are blocking. When process i returns from the call* MPI_*A the operation has been completed from the point of view of process i. For data exchange or reduction collectives all data in the send buffer have been sent, and data or results from other processes have been received in the receive buffer.*

According to Definition 1 collective operations are *not* synchronizing – with of course one notable exception, namely the explicit MPI_Barrier synchronization operation. Completion of a collective by process i does not imply anything about what has happened for the other processes j in the set of processes performing the collective operation. It may even be possible

for process i to complete a collective operation before any of the other processes have even performed the call!

Note also that by Definition 1 collective operations are blocking. Process i returns from the collective call when the operation has been completed from this process' point of view. A collective *progress rule* would state that a collective call would eventually complete for process i, provided that all other processes in the communicator eventually call the corresponding collective. Progress for collectives can obviously be implemented by polling, and do not require a dedicated progress thread.

The data exchange collectives come in *regular* and *irregular* variants. In the regular collectives the amount (and structure) of data sent to each of the other processes is the same, and equal to the amount (and structure) of data received from each of the other processes. In the irregular collectives, each process may send and receive a different amount of data from each of the other processes, although (with the exception of MPI_Alltoallw) the structure of the sent data, and the structure of the received data is the same for all processes j. For all data exchange collectives process i also exchanges data with itself (unless prevented by the special send buffer argument `MPI_IN_PLACE`).

The *matching rule* for the collectives is stricter than for point-to-point communication. If data are to be communicated from processes i to j, the *type signature* of the send buffer of process i must be identical to the type signature of the receive buffer on process j. In particular the *same* number of basic elements must be specified by the two processes. However, the *type maps* (layout or structure of the data) need not be identical on the involved processes.

The collectives are either *rooted* or *non-rooted* (or *symmetric*). In the rooted collectives a designated *root* process `root` plays the role of either source or destination of data or result, while in the non-rooted collectives all processes play symmetric roles.

The 16 MPI collectives are described below. As usual buffer arguments, `buffer`, `sendbuf`, and `recvbuf` are triples of the form `[buffer,count,datatype]`.

- MPI_Barrier(`comm`) synchronizes the processes in `comm`. The call by process i will not return before all other processes in `comm` have performed the corresponding MPI_Barrier call.

- MPI_Bcast(`buffer,...,root,comm`) broadcasts data placed in `[buffer,count,datatype]` on process `root` to `[buffer,count,datatype]` on all other processes in `comm`.

- MPI_Gather(`sendbuf,...,recvbuf,...,root,comm`) gathers data from all processes into `sendbuf` on the `root` process. The data in `[sendbuf,sendcount,sendtype]` from process i are placed at displacement $i \cdot$ `recvcount` $\cdot$ extent(`recvtype`) in `recvbuf` at the root process. The collective is *regular*, and each `[sendbuf,sendcount,sendtype]` must match [`recvbuf` $+ i \cdot$ `recvcount` $\cdot$ extent(`recvtype`), `recvcount`, `recvtype`] at the root process.

- MPI_Gatherv(sendbuf,...,recvbuf,recvdisp,recvcounts,...,root,comm) gathers data from all processes into sendbuf on the root process. The data in [sendbuf,sendcount,sendtype] from process i is placed at displacement recvdisp$[i] \cdot$ extent(recvtype) in recvbuf at the root process. This is the *irregular* version of MPI_Gather, and [sendbuf,sendcount,sendtype] at process i must match [recvbuf + recvdisp$[i] \cdot$ extent(recvtype), recvcounts$[i]$, recvtype] at the root process.

- MPI_Scatter(sendbuf,...,recvbuf,...,root,comm) is the opposite of MPI_Gather, with similar matching requirements.

- MPI_Scatterv(sendbuf,senddisp,...,recvbuf,...,root,comm) is the opposite of MPI_Gatherv, with similar matching requirements.

- MPI_Allgather(sendbuf,...,recvbuf,...,comm) gathers data [sendbuf,sendcount,sendtype] from all processes and stores them in [recvbuf,$p \cdot$ recvcount, recvtype] at all processes. The data contributed from process i are placed at displacement $i \cdot$ recvcount $\cdot$ extent(recvtype) in the recvbuf at each process. The collective is *regular*, and is also known as *all-to-all broadcast*, *broadcast-to-all*, *catenation* or *gossipping*. The result is as if p operations MPI_Gather(sendbuf,...,recvbuf,...,i,comm) had been performed for $0 \leq i < p$.

- MPI_Allgatherv(sendbuf,...,recvbuf,recvdisp,recvcounts,...,comm) is the *irregular* version of MPI_Allgather. Data are placed at displacement recvdisp$[i] \cdot$ extent(recvtype) in the recvbuf at each process, and [sendbuf,sendcount,sendtype] of process i and [recvbuf + recvdisp$[i] \cdot$ extent(recvtype),recvcounts$[i]$, recvtype] must match.

- MPI_Alltoall(sendbuf,...,recvbuf,...,comm) is a personalized exchange between all p processes in comm. Process i sends data [sendbuf + $j \cdot$ sendcount $\cdot$ extent(sendtype), sendcount, sendtype] to process j, and receives data [recvbuf + $j \cdot$ recvcount $\cdot$ extent(recvtype), recvcount, recvtype] from process j. Type signatures must match.

- MPI_Alltoallv(sendbuf,senddisp,sendcounts,...,recvbuf,recvdisp,-recvcounts,...,comm) is an irregular version of MPI_Alltoall. The data sent from process i to j are [sendbuf + senddisp$[j] \cdot$ extent(sendtype), sendcounts$[j]$, sendtype] and data received from process j are [recvbuf + recvdisp$[j] \cdot$ extent(recvtype), recvcounts$[j]$, recvtype].

- MPI_Alltoallw(sendbuf,senddisp,sendcounts,sendtypes,recvbuf,recvdisp,-recvcounts,recvtypes,comm) is another irregular version of MPI_Alltoall in which also the types of data sent to or received from processes j and k can be different. The data sent from process i to j are

[sendbuf + senddisp[j], sendcounts[j], sendtypes[j]] and data received from process j are [recvbuf + recvdisp[j], recvcounts[j]recvtypes[j]].

- MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm) performs a *reduction* on the data in [sendbuf,count,datatype] from all p processes using the binary operator op. The result is stored in [recvbuf,recvcount,datatype] at the root process. Operators and semantics are explained below.

- MPI_Allreduce(sendbuf,recvbuf,...,op,comm) performs a reduction on data from all p processes and stores the result in [recvbuf,count,datatype] on all processes.

- MPI_Reduce_scatter(sendbuf,recvbuf,recvcounts,datatype,op,comm) performs a reduction on data [sendbuf,count,datatype] from all p processes with count $= \sum_{i=0}^{p-1}$ recvcounts[i], and stores the part of the result starting at displacement $(\sum_{j=0}^{i-1}$ recvcounts[j]$) \cdot$ extent(datatype) in [recvbuf, recvcounts[i], datatype] of process i.

- MPI_Scan(sendbuf,recvbuf,...,op,comm) performs an *inclusive scan* of the data in [sendbuf,count,datatype] and stores the result for each process i in [recvbuf,count,datatype].

- MPI_Exscan(sendbuf,recvbuf,...,op,comm) performs an *exclusive scan* of the data in [sendbuf,count,datatype] and stores the result for each process i in [recvbuf,count,datatype].

The collectives are also defined for inter-communicators in a more or less natural way: a collective operation is performed on both subcommunicators, and the results are exchanged between the two. This extension will be of no further concern here.

For the reduction collectives, MPI defines a set of binary operators for computation of global sums (MPI_SUM), products (MPI_PROD), minimum and maximum (MPI_MIN, MPI_MAX, MPI_MINLOC, MPI_MAXLOC), and logical and bitwise values (MPI_BAND, MPI_BOR, MPI_BXOR, MPI_LAND, MPI_LOR, MPI_LXOR). These built-in operators are mathematically associative and commutative. It is also possible for the user to define own binary operators, that must be associative but not necessarily commutative. All operators operate elementwise on their input vectors. Reductions are performed in rank (canonical) order.

MPI recommends that reduction collectives are implemented such that the same result is obtained whenever the collective is called with the same arguments in the same order. It is also recommended that the elements of the input vectors are reduced the same way. This constrains the possible implementations of the reduction collectives.

Let x_i be the input vector stored in [sendbuf,count,datatype] of process i, and let $\otimes$ be either a pre- or user-defined associative, binary operator. The result of a global reduction is $y = \otimes_{j=0}^{p-1} x_j$. The result for process i of the *inclusive scan* is $y_i = \otimes_{j=0}^{i} x_j$ and of the *exclusive scan* $y_i = \otimes_{j=0}^{i-1} x_j$ for $i > 0$. The problem of computing all (inclusive or exclusive) scans for $i = 0, \ldots, p-1$ is often called the *parallel prefix* problem. Note that the inclusive

scan can trivially be computed from the exclusive scan by adding x_i locally, but the other way round is possible only if the operator $\otimes$ has an inverse. MPI originally defined only an inclusive scan operation. The MPI_Exscan collective was an MPI-2 addition.

Because of some skepticism about their efficiency, but also simply because of lack of user awareness about their existence, the collectives have not always been exploited as much as they should in MPI applications. This is changing, also because of the enormous amount of work on optimizations and much improved implementations of the MPI collectives in recent years (see Chapter 9 for some references). An early plea for efficient collective communication was pronounced in [MPSvdG95]. A not very convincing argument for using the collective communication model as the sole MPI programming model can be found in [Gor04], but for application that fit a BSP like style the collectives are convenient (MPI_Alltoall, MPI_Alltoallv, and MPI_Alltoallw immediately implement h-relations) and provides a lot of additional flexibility. An early survey of the use of collectives (as well as all other MPI functions) by frequency counting of actual applications was done in [Rab99b, Rab99a], and shows for instance that reduction collectives were already then much in demand. More recent studies of the use of collectives in scientific applications can be found in [VM03, KSOS05], and show that in particular the symmetric MPI_Allreduce is in demand, often only with small input sizes.

3.5.1 Other collective operations

Other operations in MPI are collective in the sense of Definition 1 in that all processes in the communicator argument must partake in the call. Important to mention here are the operations for explicitly creating and destroying communicators, namely MPI_Comm_dup, MPI_Comm_create, MPI_Comm_split, and MPI_Comm_free.

Also operations that implicitly create new communicators are collective. Among these are the process topology functions MPI_Graph_create and MPI_Cart_create (see Section 3.6), and the dynamic process creating functions (see Section 3.8). Other collective operations are the functions for creating one-sided communication windows (see Section 3.4) and a number of operations on files (see Section 3.7).

The MPI operations with collective semantics are summarized in Table 3.2.

3.6 Process topologies

MPI is functionally a homogeneous model in that any process can communicate with any other process in either of the three communication models and in any communicator. MPI makes no assumptions about the capabilities of the underlying hardware and communication network architecture, apart from the requirement that reliable point-to-point communication must be provided. The underlying network communication capabilities and topology are thus not reflected at the MPI level, and also essentially cannot be queried from the application (apart from weak environmental functions like MPI_Get_Processor_name). Within the

Initialization
MPI_Init
MPI_Finalize
Synchronization
MPI_Barrier
Rooted data exchange
MPI_Bcast
MPI_Gather
MPI_Gatherv
MPI_Scatter
MPI_Scatterv
Symmetric data exchange
MPI_Allgather
MPI_Allgatherv
MPI_Alltoall
MPI_Alltoallv
MPI_Alltoallw
Rooted reduction
MPI_Reduce
(Pseudo)Symmetric reduction
MPI_Allreduce
MPI_Reduce_scatter
MPI_Scan
MPI_Exscan
Communicator manipulation
MPI_Comm_dup
MPI_Comm_create
MPI_Comm_split
MPI_Intercomm_merge
MPI_Intercomm_create
Topology manipulation
MPI_Cart_create
MPI_Graph_create

One sided communication
MPI_Win_create
MPI_Win_free
MPI_Win_fence
Parallel I/O
MPI_File_open
MPI_File_close
MPI_File_sync
MPI_File_set_view
MPI_File_set_size
MPI_File_preallocate
MPI_File_set_atomicity
MPI_File_seek_shared
MPI_File_read_shared
MPI_File_write_shared
MPI_File_read_ordered
MPI_File_write_ordered
MPI_File_read_ordered_begin
MPI_File_write_ordered_begin
MPI_File_read_ordered_end
MPI_File_write_ordered_end
MPI_File_write_all
MPI_File_read_all
MPI_File_write_at_all
MPI_File_read_at_all
MPI_File_write_all_begin
MPI_File_read_all_begin
MPI_File_write_at_all_begin
MPI_File_read_at_all_begin
MPI_File_write_all_end
MPI_File_read_all_end
MPI_File_write_at_all_end
MPI_File_read_at_all_end
Process management
MPI_Comm_spawn
MPI_Comm_spawn_multiple
MPI_Comm_accept
MPI_Comm_connect
MPI_Comm_disconnect

Table 3.2: MPI functions with collective semantics.

three communication models of MPI it is therefore not possible to adapt the communication pattern of an application to the capabilities and limitations of the underlying system.

To address this limitation, the MPI standard provides a *virtual process topology* mechanism. This mechanism allows the application to describe its communication pattern or *virtual topology* to the MPI implementation, which can in turn use this information to reorder the MPI processes to make communication following this pattern more efficient. In concrete terms, this follows by creation of a new communicator in which processes of the calling communicator that have been designated as communicating with each other can have been mapped to processors that are close to each other on the physical machine.

The MPI mechanism allows description of communication patterns either as an *(un)-directed communication graph*, or as a *Cartesian grid (torus)* of arbitrary dimension, the latter virtual topology including mesh, torus and hypercubic communication patterns as special cases. Two collective MPI calls, MPI_Graph_create and MPI_Cart_create, respectively, create a new communicator in which processes (in the calling communicator) which are neighbors in the virtual topology may have been placed physically close to each other. Since the remapping is done by the creation of a new MPI communicator, communication in the new communicator is in no way restricted, say, to processes which are neighbors in the virtual topology – any pair of processes can communicate and collectives can be used – but communication between neighbors is favored, and communication between non-neighbors may be less efficient than between neighbors. In case a reordering has taken place, data from processes in the old communicator may have to be transmitted to wherever these processes have been mapped to in the new communicator. More precisely, if the rank of some process i of the new communicator is i' in the old communicator in which the virtual topology was specified, the old rank i will have to send data to process i' using the old communicator for this communication. MPI has the required functions for translating ranks between communicators, but this can be tedious.

MPI does not guarantee that communication between neighbors in the virtual topology is actually faster in the new communicator than in the old, and an MPI implementation is actually free not to do any reordering at all. A valid implementation of the topology mechanism may thus be restricted to returning an arbitrary (sub)communicator consisting of those processes mentioned in the virtual topology. In many cases this could be simply a duplicate of the calling communicator.

For both graph and Cartesian virtual topologies all processes provide the full description of the virtual topology to the MPI_Graph_create and MPI_Cart_create calls. For graph topologies this is unfortunate, leading to an inconvenient, non-scalable mechanism. First, processes may not know the intended communication of other processes, and thus additional collective calls may be required to construct the full communication graph on each process. Second, for communicationally dense applications this description will be $\Theta(p^2)$ per process, for a total (distributed) space consumption of $\Theta(p^3)$. This, as well as many other shortcomings of the MPI process topology mechanism, is discussed in more detail in Chapter 10.

3.7 Parallel I/O

Although not decidedly message passing, MPI includes a model for parallel I/O to facilitate what could be called *external communication* with files. This model provides support for partitioning of files among processes, collective operations for transferring global data structures between process local memory and files, asynchronous transfers, strided accesses, and control over physical layout. By incorporating these features into MPI this is achieved in a portable manner. The basic feature of the MPI parallel I/O model is that all data partitioning, file accessing and data structuring is expressed via derived datatypes. Mainly for this reason the parallel I/O model is of concern here. A prerequisite for efficient implementation of this part of the standard is flexible and efficient functionality for handling derived datatypes and accessing structured data.

The unit of access to and positioning within a file is called the *elementary type* (*etype*), and can be either a basic or a user-defined, derived datatype. The unit of partitioning of the file is called the *filetype* which must be constructed from the elementary type. The type maps of both elementary types and filetypes must be monotonically non-decreasing (with some further constraints on filetypes).

The MPI model defines local, blocking and non-blocking and collective operations on files. The opening of a file is a collective operation, which returns a distributed object called a *file handle* which is used for all subsequent file operations. Collective operations are thus collective over the file handle, more precisely over a hidden duplicate of the communicator used in the open call, in the sense of Definition 1.

A new feature not found elsewhere in MPI is a restricted type of non-blocking collective file operation, called *split collectives*. Split collectives consist of a collective MPI_File_*A*_begin call, which initiates the operation, and a collective MPI_File_*A*_end call, which completes the operation. They were introduced to provide means to overlap the high latency of access to file storage with useful computations and communications. The split collective feature is limited in that only one split collective operation can be in progress on at file handle at the same time, in that no other collective file operation can be called as long as a split collective is in progress, and in that a split collective operation called on one process does not match a semantically equivalent, normal collective operation called on another process of the file handle. With the same motivation of providing for overlap between collective operations and computation, it was considered to include also non-blocking, split collectives for synchronization, data exchange and reduction into the MPI standard. This was eventually rejected, see the MPI Journal of Development, `www.mpi-forum.org`. A different, more natural proposal for *non-blocking collectives* that is analogous to the non-blocking point-to-point operations (using a request object for ensuring completion) is likely to become included in MPI 3.0 [HKG+07].

3.8 Dynamic process management

The dynamic process management facility makes it possible for a set of MPI processes to collectively spawn a new batch of programs in parallel. This new set of programs can again

initialize MPI (that is, must call MPI_Init), which will give them their own MPI_COMM_WORLD communicator. Since communicators in MPI are static objects, there will (and can) be no change in the already running set of MPI processes. To facilitate communication between the set of spawning and the set of spawned processes, a new *inter-communicator* is created having the spawning communicator and the MPI_COMM_WORLD of the spawned processes as sub-communicators. This inter-communicator is returned by the spawn call at the spawning processes. At the spawned processes it is an implicitly existing *parent* communicator, that can be accessed through the MPI function MPI_Comm_get_parent.

The spawning facility makes it possible to spawn processes in the SPMD paradigm (copies of the same program), as well as to start different programs (MPMD paradigm).

The process management facility in addition provides means for different, already running MPI processes to establish communication. This is perhaps one of the less well thought out parts of the MPI-2 additions to MPI.

The dynamic process management facilities are of relevance here only in so far as many of the operations are collective, and thus of concern for the verification interface described in Chapter 11

3.9 Assessment of the standard

A famous quote says that MPI was designed "*not to make easy things easy, but to make difficult things possible*" (William D. Gropp, EuroPVM/MPI 2004, "MPI and High Productivity Programming"). This is surely part of the reason why MPI has, since it was introduced in 1994, been such an overwhelmingly successful standard. More concrete reasons for the acceptance of MPI among both application scientists and parallel processing researchers include the following.

- MPI was the right interface at the right time. It adopted the right ideas from the various message passing frameworks around in the early 1990ties, and therefore gained acceptance fast. It helped tremendously that MPI was closely followed by a template implementation that was already in its first versions of good quality [GLDS96].
- MPI was quickly adopted by the parallel processing and computer science community, and it was possible to publish findings and results related to MPI. A dedicated message passing user group conference, EuroPVM/MPI, was initiated already in 1994.
- Economy of concepts. Despite being a large standard, MPI is centered around a small number of key concepts, and it is not too difficult to start using the standard. Mastering the standard, as with all good things, takes time and experience.
- MPI is a well thought out and well engineered standard. The key concepts fit and complement each other, and the number of conceptual mistakes is small.
- MPI is universal, and indeed makes easy things more or less easy, and difficult things possible.

- MPI supports building of higher level, application specific libraries, which can hide much of the more tedious, MPI low level communication.
- MPI has allowed (efficient) implementations on a wide variety of systems.
- MPI can live together with many other parallel interfaces, for examples OpenMP for explicit, hybrid, shared-distributed memory programming (it is not discussed here if and when this approach is advantageous, but there has been a number of studies on the pro and cons of explicit hybrid programming, see for instance [CE00, Hen00, Rab02, SS01, SSOB02, SSOB03]).
- MPI has turned out to be scalable, both as a programming model and as an implementation to thousands (Earth Simulator and other systems) and even hundred of thousands processors (Blue Gene/L).
- The standard has few optionals and exceptions, and though informal, is quite precise (except where deliberately left open to different interpretations). There is in principle no question about what MPI is. There are only few query functions in the standard for an application to determine the level of functionality supported.
- MPI has supported the construction of necessary performance analysis tools.
- MPI has been and continues to be an interesting and worthwhile topic to work on.

Part II

Towards efficient Implementations of MPI

Chapter 4

Technical motivation and framework

This part summarizes the technical contributions of the dissertation, documented by the 26 technical papers collected in Part III. The driving motivation has in all cases been concrete problems arising in the work with NEC's proprietary MPI/SX and MPI/ES implementations for the NEC SX vector computers and the Earth Simulator, or in the translation of results and experiences from MPI/SX into the MPI/EX and MPI/PC implementations for various scalar NEC systems. The papers were all written hand in hand with development and implementation of proprietary, commercial software, which has a few noteworthy consequences.

- The described algorithms have all been implemented and *tested*. As with all software that is used in production and beyond the control of the implementer bugs have been and continue to be found and fixed. There is often a significant amount of experimentation and tuning which has not found mention in the papers, partly for reasons of propriety.

- Commercial grade software, especially fundamental libraries like MPI, require implementations that are correct without exceptions. This is ascertained by extensive and continued testing, as well as by feedback from users.

- Although performance (especially for the collective communication routines) is (most) often measured by synthetic benchmarks, eventually performance in the context of actual user applications is what matters for a commercial, application support library, and sometimes tunings that may be detrimental in some isolated benchmark have been performed based on user feedback and application codes.

- Since the implementations belong to NEC proprietary software, source code is not available for public inspection or use. In some cases specific implementation details, that could have been of interest, are not revealed in the papers.

- Both algorithms and concrete implementations as described in the papers constitute a snapshot of a continuous and ongoing development history. Many further refinements are not described in a publicly accessible form (but are sometimes touched upon in the

following). Often, but not always, this kind of engineering work is not interesting to a wider audience.

The technical work can be broadly characterized as *parallel algorithm engineering*, spanning aspects of algorithm development, analysis and implementation, algorithm refinement and experimentation, software engineering, and library and interface design.

Experiments have been performed on the rather limited range of parallel computer systems that has been available. Basic development has been done on NEC SX systems, rarely with more than 4 processing nodes (see Chapter 5). Scalability studies, experiments and benchmarking have mostly been done on a 32 node, dual-processor AMD based PC-cluster with Myrinet interconnect, and on a 16 node, four processor Itanium system with Infiniband interconnect. In todays landscape of parallel processing where systems programmed with MPI routinely have hundreds or thousands (and sometimes more) of processing nodes this is of course hardly defensible. During some hectic weeks of installation and testing of MPI/ES for the 640 node Earth Simulator in February 2002 basic debugging and benchmarking on a larger system was possible, but unfortunately with little further opportunity for systematic experimentation and benchmarking.

The technical contributions cover most aspects of the MPI standard, excluding point-to-point communication and process management, with particular emphasis on efficient datatype handling and parallel I/O (Chapter 7), one-sided (Chapter 8) and collective (Chapter 9) communication, process topologies (Chapter 10), and means for ascertaining correctness and performance (Chapter 11).

Chapter 5

Target architectures

MPI/SX is NEC's proprietary MPI implementation for the SX series of *parallel vector supercomputers*, and this is the primary target architecture for the algorithms and implementations described in the following chapters. The NEC SX vector supercomputers are all, starting from the SX-4, *multi-node* architectures, consisting of a number of SMP nodes which are interconnected by a powerful, NEC proprietary switch, called the IXS (*internode crossbar switch*). The processors on the nodes are *vector processors*.

A vector processor is characterized by the ability to perform arithmetic and logical operations on vectors of elements, and by the corresponding high memory bandwidth, see [PH96, Appendix B] and [HSN81, Ble90]. Despite the extreme increase in peak processor performance by scalar processors in the last decades (which is now flattening out), vector processors can still be an order of magnitude faster, and also the sustained performance achievable with vector processors tends to be very high, provided a significant fraction of the code lends itself to *vectorization*. This is often the case for "scientific codes", see for instance [OCS+03, OCC+04, OCC+05]. Vector processors are expensive because of the heavily banked memory required to sustain the memory bandwidth required to keep the vector pipes busy. Currently only NEC is left as a vendor of high-performance vector systems. Convex and other American vendors have left the market long ago, Fujitsu and Hitachi are no longer producing vector (or vector-like) systems, and it seems to be generally agreed that the Cray X1 somehow did not meet the expectations, although Cray claims to remain committed to vector systems (see `www.cray.com`).

The SX processor is a traditional, but very mature vector processor. It supports vectorized arithmetic and logical operations on consecutive and strided vectors of many types, very efficient gather-scatter operations, and horizontal operations like sum, prefix, min-max, pop-count, and search. All vector operations operate directly on main memory and there is no vector cache. The relatively slow scalar processor is cache-based.

In the current SX-6 and SX-8 systems (up to) 8 vector processors are connected to a shared memory, and treated as an SMP node. For performance reasons the system is not cache-coherent, and various techniques must be used to deal with this. Importantly, vector instructions operate directly on memory and circumvent the cache, and the architecture provide a number of memory fence and atomic operations. The memory model is a relaxed

consistency model [PH96, CSG99]. The nodes and processors run under an NEC proprietary UNIX variant called SUPER-UX. In addition to MPI, the SMP nodes can be programmed in OpenMP and other shared memory paradigms. Under SUPER-UX processes cannot directly access the *process local memory* of other processes, which makes direct communication via shared memory impossible. Instead, the operating system provides a facility for allocating special, non-swappable *global memory* outside of process local memory space, and that can be accessed by other processes both on the same an on other nodes. This feature is extensively used in MPI/SX for intra- and inter-node communication.

Multiple SX nodes can be coupled by a powerful, single-stage crossbar, called the IXS. Data can be transferred in consecutive (and with limitations, strided) blocks by asynchronous transfer operations between global memory segments of the involved nodes, that are mapped into a global address space. The IXS also supports a limited number of atomic operations, and since the SX-6 a small set of special counter registers that can be used for implementing a constant time barrier operation (independent of the number of nodes). The IXS supports bidirectional, send-receive communication, which means that a node can send data to a node and at the same time receive data from another, possibly different node.

SX systems typically have from 4 to 24 nodes. The currently largest SX system in Europe is the 72-node SX-8 system at the *Hochleistungsrechenzemtrum* (HLRS) in Stuttgart, Germany (see `www.hlrs.de`).

The SX architecture was the basis for the *Earth Simulator*, a 640 node parallel vector system based on the SX-6 system with a more powerful memory and switch (called the IN switch). The Earth Simulator was designed especially for large scale climate research and Earth system modeling, is installed at the *Earth Simulator Center* in Yokohama, Japan, by the national *Japan Agency for Marine-Earth Science and Technology* (JAMSTEC), and was inaugurated in April 2002. At that time the machine caused a tremendous uproar by being significantly faster and more capable than the till then dominating US based systems by IBM, Cray and other American vendors. Even in the popular conception of things, the US establishment was challenged, as witnessed by almost hysterical articles also in serious media like the *New York Times* that spoke of a "Computenik" (in analogy to the uproar caused by "Sputnik" in 1957) [Mar02]. What actually drove these outbreaks of emotion is obscure, but in any event the emergence of the Earth Simulator was used to inject money and new energy into – *invigorate* – the US high-performance computing community and industry. In that sense the Earth Simulator had a positive and lasting effect. The Earth Simulator immediately entered the No. 1 position of the Top 500 list (see `www.top500.org`), the dubious, much hype-generating ranking of high-performance systems based on the Linpack benchmark, and stayed there for an unusually long period of two and a half years. It was eventually superseded in 2005 by the IBM Blue Gene/L.

Technical overviews of hardware and system software for the Earth Simulator can be found in a Special Issue of Parallel Computing from 2004 [ZS04] and elsewhere [YHK+99]. That the Earth Simulator was also a success for its intended application areas is perhaps witnessed by the several Gordon Bell awards[1] it won in the years after it became opera-

[1] Awarded annually at the ACM/IEEE Supercomputing conference for highest application performance.

Model	Year	Peak/Proc	Mem BW/proc	Mem/node	n	N	p	IXS BW/node
SX-4	1995	2GFLOPS	16GByte/s	16GByte	32	16	512	8Gbyte/s
SX-5	1997	8GFLOPS	64GByte/s	128GByte	16	32	512	8GByte/s
SX-6	2001	8GFLOPS	32GByte/s	64GByte	8	128	1024	8GByte/s
ES	2002	8GFLOPS	32GByte/s	16GByte	8	640	5120	12.3GByte/s
SX-8	2005	16GFLOPS	64GByte/s	128GByte	8	512	4096	16Gbyte/s

Table 5.1: System capabilities of the NEC SX series and the Earth Simulator. The maximum number of processors per node is denoted by n, the maximum number of nodes by N, and the maximum number of processors by $p = nN$. The Earth Simulator is a once-in-a-lifetime system with exactly the configuration given by n, N, p.

tional [SMSY02, YIU+02, STF+02, KTJT03], and the very large number of papers in the wake on actual scientific results obtained with the aid of the system. An assessment and comparison to other systems from a high-performance computer science perspective can be found in [OCS+03, OCC+04].

Table 5.1 gives the basic performance characteristics of the SX series from SX-4 to SX-8[2]. For data sheets concerning the SX series and the Earth simulator, see `www.hpce.nec.com` and `www.es.jamstec.go.jp`.

High level design concerns for an MPI library for the SX computers include efficient vectorization, especially of all bulk transfers of data, short critical paths of scalar instructions before vector instructions, efficient use of the asynchronous communication facilities of the IXS, use of the atomic registers of the IXS, and awareness of the SMP structure of the system.

Other target systems are the NEC TX-7 (AsAma) with 32-processor SMP-nodes with Intel IA64 processors, coupled with Infiniband, as well as smaller Intel IA64 or IA32, or AMD Athlon or Opteron based systems, also with either Infiniband or Myrinet as interconnect. A port of MPI/SX to a Quadrics [BAH+05] based system has also been developed.

[2]The next generation SX-9 was announced for ACM/IEEE Supercomputing 2007. It is substantially more powerful than the SX-8, and has a substantially different interconnect.

Chapter 6

Overview of MPI/SX

MPI/SX is the MPI implementation for NEC's SX series of parallel vector computers, and the basis for other developments like MPI/ES for the Earth Simulator, MPI/PC for Myrinet-based PC-architectures, and MPI/EX for the IA64-based NEC TX7 (AsAmA) with Infiniband interconnect.

The development of MPI/SX was started in 1997, and was originally based on the original `mpich` implementation from Argonne National Laboratories [GLDS96]. An overview of the first MPI/SX implementation can be found in [HRZ98]. MPI/SX has since then deviated significantly from the `mpich` framework, especially through the requirements posed by the MPI-2 standard (parallel I/O, one-sided communication, and process management). In October 2000 MPI/SX was arguably the first implementation to support the full MPI-2 standard without any restrictions. The only contender at that time, the Fujitsu implementation from 1999, described in [AKL99, BHS+02], was lacking in process management (particularly the MPI_Comm_Accept/MPI_Comm_Connect functionality), and that implementation has since then apparently not been developed further. MPI/SX was also the basis for MPI/ES for the Earth Simulator, and all development was done by the NEC C&C Research Laboratories (by Hubert Ritzdorf and the author).

A general overview of the design and performance of MPI/SX on first the NEC SX-5, later the SX-6 and most recently the SX-8 of MPI/SX is given in the two papers:

1. Maciej Gołebiewski, Hubert Ritzdorf, Jesper Larsson Träff, and Falk Zimmermann. The MPI/SX implementation of MPI for NEC's SX-6 and other NEC platforms. *NEC Research & Development*, 44(1):69–74, 2003.

2. Hubert Ritzdorf and Jesper Larsson Träff. Collective operations in NEC's high-performance MPI libraries. In *International Parallel and Distributed Processing Symposium (IPDPS 2006)*, page 100, 2006.

Other full MPI-2 implementations have followed, notably `mpich2` and OpenMPI [GFB+04, SL04], and several other vendor implementations which implement at least significant parts of the MPI-2 extensions.

A main feature of the MPI/SX implementations is its highly tuned point-to-point communication. The implementation uses many standard techniques but highly tuned and adopted to the SX vector architecture. Obviously, all copying of user data is done by vectorized copy operations.

The point-to-point communication implementation (by Hubert Ritzdorf) uses a sophisticated implementation of the socalled lock-free technique, in which each MPI process has a small, constant number of *slots* for each other MPI process, into which header information and small data loads (up to about 1KBytes) can be written. Bookkeeping and acknowledgments for keeping track of free slots is done by piggybacking on other messages as far as possible to save traffic. To achieve best performance, as is common, three protocols are used, socalled *short*, in which a message is written directly into a slot, *eager* in which an intermediate, global memory buffer is allocated and later read and freed by the receiving process, and *long* or *rendezvous*, in which pipelining with several blocks is employed. If user data has been placed in global memory, communication can in many cases be done by a direct memory copy or transfer operation. In this case a modified eager protocol is used. Unexpected eager messages give rise to allocation of an intermediate buffer by the receiving process. The MPI/SX implementation is therefore relatively memory consuming and could run into scalability problems for very large systems. For current SX systems and also for the Earth Simulator with their huge memories this has so far not been problematic.

Tuning of the point-to-point communication is done with application and not benchmark performance in mind. Nevertheless, MPI/SX achieves more than 90% of the peak bandwidth both for intra-node communication (MPI processes on same SMP node) and inter-node communication (MPI processes on different SMP nodes) on point-to-point benchmarks. Recent point-to-point performance is given in Table 6.1, taken from [2].

MPI provides the special memory allocator function MPI_Alloc_mem. In MPI/SX this allocates a buffer in non-swappable, global memory. A corresponding memory allocator has been implemented to support this. For user buffers allocated with MPI_Alloc_mem all intra-node communication can be done by a direct memory copy operation, irrespective of whether the user data are consecutive or not. Inter-node communication by a direct transfer operation is possible if data are consecutive and aligned. If these conditions are not met, the communication is done by the protocols described above, using intermediate global memory buffers. As seen from Table 6.1, provided that the amount of data transferred is large enough, a very high fraction of peak bandwidth is also achieved for buffers that are not allocated in global memory. This is due to pipelining, by which the time spent in copying into and out of the intermediate pipelining buffers is effectively hidden. This is an advantage of the point-to-point communication model: both sending and receiving process are involved in the communication.

MPI/SX is a non-threaded implementation of MPI. Slots are checked at each MPI communication operation, and progress is thus only guaranteed if the application regularly performs MPI communication. MPI/SX has a highly efficient, vectorized implementation of this progress machinery.

Being a high-performance implementation, only error checking that can be done locally

MPI/SX on SX-8

Communication	Buffer	Latency	Bandwidth	Peak
Intra	local memory	$1.6\mu s$	28.1 GBytes/s	32 GBytes/s
	global memory	$1.6\mu s$	30.7 GBytes/s	32 GBytes/s
Inter	local memory	$4.7\mu s$	11.7 GBytes/s	16 GBytes/s
	global memory	$4.7\mu s$	15.0 GBytes/s	16 GBytes/s

MPI/ES on the Earth Simulator

Communication	Buffer	Latency	Bandwidth	Peak
Intra	local memory	$2.0\mu s$	14.4 GBytes/s	16 GBytes/s
	global memory	$2.0\mu s$	15.8 GBytes/s	16 GBytes/s
Inter	local memory	$5.2\mu s$	9.3 GBytes/s	12.3 GBytes/s
	global memory	$5.2\mu s$	12.2 GBytes/s	12.3 GBytes/s

Table 6.1: Communication bandwidth and latency for intra- and inter-node communication with MPI/SX and MPI/ES on the SX-8 and the Earth Simulator. Communication buffers are placed either in MPI process local memory or in special, non-swappable global memory by the MPI_Alloc_mem call.

is performed in MPI/SX (with a few exceptions for parallel I/O).

Point-to-point communication operations are made available internally in more efficient versions (no error checking, no lookup or dereferencing for MPI handles) for use in collective algorithms, parallel I/O and one-sided implementations. These internal ADI (*Abstract Device Interface*) functions mostly have the same functionality as the corresponding MPI functions, but are extended with special send and receive functions with completion counting (see Chapter 8) and overlapping of function execution and communication (see Chapter 9).

A number of features and special optimizations are not documented in technical papers. Among these is a linear time, vectorized implementation of the MPI group management functionality.

Chapter 7

Efficient handling of derived datatypes

Early MPI implementations like MPICH [GLDS96] employed a straightforward handling of non-consecutive user data described by derived datatypes. Sending non-consecutive data typically entailed packing of the *complete* communication buffer into a consecutive, intermediate buffer before communication. Reception of structured data likewise entailed an extra unpacking step from a consecutive, intermediate communication buffer.

Packing and unpacking were done by simple, recursive algorithms, following closely the structure of the data as set up by the MPI type constructors. In particular, each repetition count in the datatype gave rise to a corresponding number of recursive calls. The number of recursive calls in this straightforward implementation was therefore unbounded in the sense of standing in no relationship to the size of the DAG describing the derived datatype. Furthermore, the pack and unpack functionality was only designed to handle full user buffers `[buffer,count,datatype]`, and therefore not suited to neither point-to-point nor to collective algorithms employing pipelining.

All this contributed to a significant overhead in the use of non-consecutive, derived datatypes, which from early on caused users to be reluctant in using MPI datatypes, even when they would be natural from a descriptive, application point of view. Similarly, these inefficiencies hampered the efficiency of the parts of MPI that intimately depend on derived datatypes, namely parallel I/O.

These problems were addressed from early on in MPI/SX. Four papers describe the improved handling of derived datatypes by a technique which was termed *flattening on the fly*, and the use of the developed mechanisms in the parallel I/O part of MPI/SX.

1. Jesper Larsson Träff, Rolf Hempel, Hubert Ritzdorf, and Falk Zimmermann. Flattening on the fly: efficient handling of MPI derived datatypes. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 6th European PVM/MPI Users' Group Meeting*, volume 1697 of *Lecture Notes in Computer Science*, pages 109–116, 1999.

2. Ralf Reussner, Jesper Larsson Träff, and Gunnar Hunzelmann. A benchmark for MPI derived datatypes. In *Recent Advances in Parallel Virtual Machine and Message*

Passing Interface. 7th European PVM/MPI Users' Group Meeting, volume 1908 of *Lecture Notes in Computer Science*, pages 10–17, 2000.

3. Joachim Worringen, Jesper Larsson Träff, and Hubert Ritzdorf. Fast parallel non-contiguous file access. In *Supercomputing*, 2003. `http://www.sc-conference.org/sc2003/tech_papers.php`.

4. Joachim Worringen, Jesper Larsson Träff, and Hubert Ritzdorf. Improving generic non-contiguous file access for MPI-IO. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 10th European PVM/MPI Users' Group Meeting*, volume 2840 of *Lecture Notes in Computer Science*, pages 309–318, 2003.

For MPI/SX the simple, recursive packing and unpacking implementation was a major problem. The simple, recursive algorithms visit the leaves of the type DAG many times (as determined by the repetition counts in the type DAG), and for each visit, only a small unit of consecutive data can be copied. This caused a large scalar overhead, and effectively blocked efficient vectorization, which works well only on larger units – which, on the other hand, do not have to be consecutive, but can be either regularly spaced by some fixed *stride*, or irregularly spaced as determined by an index vector.

A solution, which exploits the regularities inherent in MPI derived datatypes (assuming that at least some internal repetition counts in the type DAG are large) to provide much larger units of data for copying in a vectorized operation was developed in [1]. The type DAG is still traversed recursively, but each leaf is visited only once. This is achieved by building a stack of repetition counts, sizes and extents of the types encountered during the DAG descent towards each leaf. When the leaf is reached, all information required to copy all instances of the leaf are present on the stack, and can be processed by counting through the stack. To efficiently exploit vectorization, the largest repetition count c present on the stack is found, and c copies of the leaf (which forms a consecutive sequence of some b units) spaced by the *extent* associated with c are copied to/from the consecutive pack/unpack buffer spaced by the *size* associated with c. This is called the *largest count* optimization, and is important for the vector system. For MPI/SX this copy function has furthermore been written in assembly language. This is the idea termed *flattening on the fly*, a term which shall indicate that the type DAG is only implicitly flattened and the type map not constructed at all. Therefore, the technique can also be described as *listless* handling of MPI derived datatypes.

Flattening on the fly has the following property.

Property 1 *Let L be the number of leaves in the type DAG (the number of different ways a DAG leaf can be reached from the root following the MPI datatype rules, as defined in Section 3.2), and T the tree constructed as the leaves are visited. The total time for traversal of the DAG needed for packing or unpacking a buffer of m basic datatypes is $O(|T|)$.*

Compared to the simple, recursive packing and unpacking algorithms this is an improvement from something that is not bounded by the DAG size but is $O(m)$ down to $O(|T|)$. Since MPI types are very compact descriptions of data layouts, typically $|T| \ll m$.

The other problem associated with the simple, recursive datatype handling algorithms was lack of or very inefficient functionality for *partial* packing and unpacking. This made it impossible to implement efficient pipelined point-to-point and collective communication algorithms for non-consecutive data.

MPI/SX implements partial packing and unpacking ADI functions MPIR_Pack_long and MPI_Unpack_long, which packs and unpacks a certain number of units (bytes) of a triple `[buffer,count,datatype]` starting after a given number of units in the type map have been skipped. An important feature of the MPI/SX functionality is that the time to pack/unpack m units is independent of the number of units skipped.

Property 2 *Let L be the number of leaves in the type DAG, T the tree constructed as the L leaves are visited, m the number of units to be packed/unpacked, and s the number of units of the type map to be skipped. The time for the pack/unpack operation is $O(|T| + m)$, and independent of s.*

The actual constants hidden in the $O(|T| + m)$ run time depend on the amount of regularity in the datatype. If a sufficiently large repetition count is present somewhere in type DAG, and if the number of different subtypes in substructures is not too large, the packing and unpacking functions are comparable to straight memory copy operations (in the SX vectors machine). For cache-based systems, the largest count optimization can sometimes lead to bad cache behavior, and further optimizations for this case are needed. Some work in this direction is described in [BGST03, RMG03, BSTG06]. Flattening on the fly or similar techniques are used or developed further in e.g. [WWP04, SWP04]. Whether these implementations also support partial packing and unpacking efficiently is not known.

Since the publication of [1] flattening on the fly has been extended to also handle MPI indexed data types with fixed blocklengths as can be constructed with the MPI_Type_create_indexed_block constructor. Also, all implementations have been considerably refined and tuned over the years.

In [1] a batch of parametrized derived datatypes was defined as a benchmark to evaluate the flattening on the fly technique. The achieved bandwidth for each `[buffer,count,datatype]` consisting of m units was compared to the bandwidth of sending m units as a consecutive buffer, which serves as a best case baseline. For datatypes with regularities, that is MPI vector types and indexed and structured types with few component subtypes, a very high fraction of best possible performance is achieved by the flattening on the fly technique. The batch of parametrized datatypes was improved and extended in [2], and incorporated into the general MPI benchmark framework *SKaMPI* [RSPM98, RST02].

The initial MPI/SX implementation of parallel I/O was based on the ROMIO implementation from Argonne National Laboratories, see [TGL98, TGL99b, TGL99a, TGL02]. This implementation uses an explicit representation of datatypes as offset-length lists, that are both inefficient for accessing data described by the type (as described above) and costly in memory space. The deficits of this representation, especially in the context of ROMIO, are analyzed carefully in the two papers [3] and [4], which develop a listless I/O implementation. To replace the list-based datatype handling in ROMIO with the flattening on the

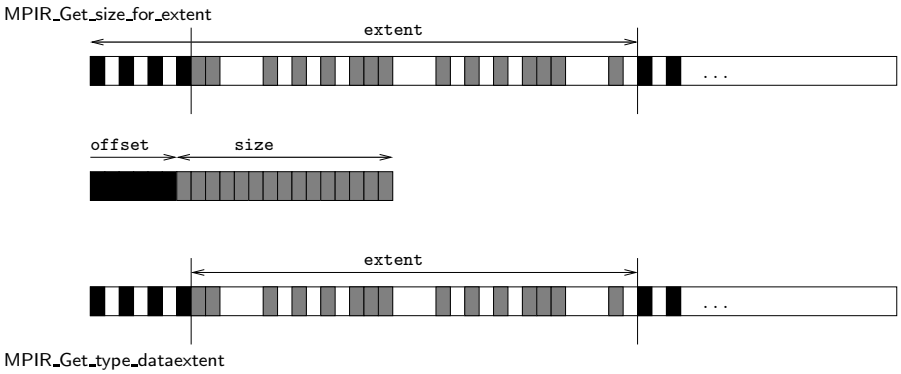

Figure 7.1: The type navigation functions for *listless* I/O.

fly technique, more functionality is needed for navigation within typed MPI buffers. For this purpose two new, loosely orthogonal functions were introduced to the MPI/SX ADI, as illustrated in Figure 7.1:

- MPIR_Get_size_for_extent(`datatype,offset,extent`) computes the total size (of basic elements) occupied by the `datatype` in a given `extent` in memory after skipping a certain number of `offset` bytes.
- MPIR_Get_type_dataextent(`datatype,size,offset`) computes the extent in memory if `size` bytes of the `datatype` are unpacked (after skipping `offset` bytes) from a consecutive buffer.

For types with L leaves with traversal trees T both of these functions take $O(|T|)$ steps with a small constant. The functions depend on the MPI restriction of elementary and filetypes to have monotonically non-decreasing type maps. For handling moderately long, intermediate buffers, the datatype handling functionality of MPI/SX therefore comes with very low overhead.

Chapter 8

One-sided communication

The one-sided communication model is one of the important additions to MPI introduced with the MPI-2 extensions [GHLL+98]. Two early papers describe the NEC implementations for MPI/SX and MPI/PC.

1. Jesper Larsson Träff, Hubert Ritzdorf, and Rolf Hempel. The implementation of MPI-2 one-sided communication for the NEC SX-5. In *Supercomputing*, 2000. `http://www.sc2000.org/proceedings/techpapr/index.htm#01`.

2. Maciej Gołebiewski and Jesper Larsson Träff. MPI-2 one-sided communications on a Giganet SMP cluster. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 8th European PVM/MPI Users' Group Meeting*, volume 2131 of *Lecture Notes in Computer Science*, pages 16–23, 2001.

From the beginning the MPI/SX implementation supported fully the MPI-2 one-sided model, including the passive, lock synchronization mechanism, and exploited many of the possibilities given with the MPI-2 standard for optimizations for the NEC SX vector systems, in particular the *assertions* `MPI_NO_PRECEDE` and `MPI_NO_SUCCEED` that can be given to the MPI_Win_fence call. Other early, less mature implementations were described in [BM00] for SUN MPI, in [MS99, BPS01] for a Windows based MPI called WMPI [MS98, PS00], and for SCI clusters in [WGR02].

The one-sided communication operations between user memory of the origin process and the communication window at the target are performed either by direct memory copy or IXS transfer, or by hidden point-to-point communication. For the former to be possible the accessed memory in the window of the target must be allocated in global, non-swappable memory, and if origin and target reside at different SX nodes, also the memory of the origin must be in global memory. Furthermore, for a direct IXS transfer to be possible, both origin and target data must be contiguous, and the buffers must be aligned (on 8-Byte boundaries). If the conditions for direct copy or IXS transfer are not fulfilled, the communication is done by the MPI/SX internal point-to-point mechanism. Allocation in global memory can be done by the user with the MPI_Alloc_mem call, which is implemented as explained in Chapter 6. The MPI_Accumulate operation in MPI/SX always maps to point-to-point communication.

For MPI/ES the capability of the IXS to transfer data asynchronously with later completion has been exploited in the one-sided model to permit overlap between communication and computation at the user-level.

Communication operations that are mapped to point-to-point communication send a header to the target process, possibly including (short) data. Since MPI/SX is non-threaded, progress of communication depends on the target process eventually performing an MPI call, upon which the request from the origin is processed. For an MPI_Put operation the data are copied either into user space (for short and eager messages), or an ADI receive request is posted to later receive the data. For an MPI_Get operation an ADI send operation is posted to facilitate the transfer to the origin process. For the implicit sending and receiving of data two new functions MPIR_Isend_count and MPIR_Irecv_count which signal completion by incrementing a counter argument were introduced into the ADI of MPI/SX. The one-sided synchronization operations can ensure completion on origin and target processes by waiting for these completion counters to reach the number of one-sided communication operations that were mapped onto point-to-point communication in the epoch being completed.

A further complication of the one-sided model is that the datatype of the data being accessed at the target is supplied by the origin, and may not be known at all at the target (here, stricter SPMD requirements to MPI programs might have helped). Datatype descriptions may therefore have to be transferred to the target from the origin along with the one-sided communication operation. For all committed datatypes, MPI/SX computes a compact, linear array representation that, if necessary, is sent as part of the header (if small enough to fit, otherwise as an eager message) to the target. Upon processing the request, the target can reconstruct the derived datatype, and use the MPI/SX datatype handling routines to further process the structured data. Such datatypes are kept at the target, and the origin marks datatypes that have been sent, so that the overhead of sending and reconstructing a derived datatype is only paid once. The ADI of MPI/SX contains functionality for type reconstruction that is also used in parallel I/O.

The dedicated MPI_Win_start-MPI_Win_complete, MPI_Win_post-MPI_Win_wait synchronization mechanism is implemented by messages as described in [GHLL+98]. Also the passive MPI_Win_lock-MPI_Win_unlock mechanism is, for the multi-node case, implemented by lock and unlock control messages. This, in connection with the absence of a progress thread, could lead to performance problems with passive, one-sided communication, but if also the target process is regularly involved in (other) MPI communication, this will not be the case. For the single-node case, where all processes in the window reside on the same SX node, and if the exposed memory is in global memory for all processes, atomic operations are used to implement the lock mechanism. Only locks that cannot be granted cause the locking process to wait for an acknowledgment message. Since atomic operations are also supported over the IXS, this mechanism will in future MPI/SX versions be extended also to the multi-node case. Other implementation alternatives for the lock mechanism are discussed in [JLJ+04].

The scalable MPI_Win_fence synchronization mechanism uses a special-purpose, regular reduce-scatter function MPI_Reduce_scatter_block, implemented as described in Section 9.4 to count the number of point-to-point mapped, one-sided communication requests that have to

be completed before the next epoch. If it can somehow be known that all one-sided communication was performed by memory copy operations or IXS transfers, a barrier synchronization suffices for an even more efficient MPI_Win_fence operation. MPI/SX has introduced a special assertion `NEC_MPI_GETPUT_ALIGNED` allowing the user to state to the library that this condition is fulfilled.

The one-sided synchronization mechanisms are also surveyed in [1], but the explanation of MPI_Win_fence is too strong: after completion all one-sided operations of which a process was origin have completed (completion on target is not necessarily guaranteed). In both papers completion counting is described as done by MPI_Allreduce. This has subsequently been improved by using the less expensive MPI_Reduce_scatter_block.

In both papers [1] and [2] the implementation is benchmarked with a simple, parametrizable neighbor exchange program. Since the resulting communication pattern can be implemented equally well by point-to-point communication with MPI_Sendrecv, this gives an opportunity to compare the two models on fair grounds. In the one-sided model, the exchange pattern is implemented using all three synchronization mechanisms, and also the synchronization overhead of these are measured in isolation. For small messages and few neighbors or blocks, the explicit synchronization overhead for one-sided communication is, as could be expected, significant, but for long data and/or many neighbors or blocks, this can be amortized, so that with this benchmark there is no very significant difference between the point-to-point and the one-sided model. The recommendation, which seems warranted, is that the MPI/SX user can choose the model which fits best with the application. There will be no significant performance penalty by the choice. Similar benchmarks were developed and used in [ASW05, GFD03, GT07b].

Implementation alternatives for the one-sided synchronization mechanisms were more systematically explored in [TGT04, GT05]. Some of the alternatives discussed there save on synchronization messages by postponing communication to the end of the epoch. This is an interesting option. In general, the one-sided model gives the implementer full freedom as to when communication should happen, thus the opportunity to group small messages for better exploitation of bandwidth is given. It could even be possible to schedule transfers. Such options have so far not been explored in the MPI/SX implementation, and likely also not in any other one-sided implementations. Such optimizations may – for long, communication intensive epochs – also have drawbacks by leading to more network contention at the end of the epoch. It would be interesting to explore these alternatives by more systematic benchmarking. This has so far not been done.

Chapter 9

Collective communication

From the outset MPI/SX has emphasized efficient, scalable implementations of the MPI collectives. This implies both *efficient implementations* targeted to the SX vector architecture, and the IXS interconnect of the multi-node SX systems, as well as asymptotically *efficient algorithms* under communication and computation models that reasonably abstract the capabilities of the SX hardware (for example homogeneous, bidirectional, send-receive communication between nodes). For the SX-4 only the single-node case was important, and early shared-memory algorithms were described in [HRZ98]. Shared-memory algorithms and implementations have since then been significantly refined and improved, mostly using standard techniques [MCS91, SvdVL99], and this work is not further described here. From the SX-5 and onwards collectives for the multi-node case have gained in importance, culminating in the 640 nodes of the Earth Simulator, although single-node performance continues to be important. The papers of this chapter deal primarily with collective algorithms and optimization for the multi-node case.

1. Jesper Larsson Träff. A simple work-optimal broadcast algorithm for message passing parallel systems. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 173–180, 2004.

2. Jesper Larsson Träff and Andreas Ripke. Optimal broadcast for fully connected networks. In *High Performance Computing and Communications (HPCC'05)*, volume 3726 of *Lecture Notes in Computer Science*, pages 45–56, 2005.

3. Jesper Larsson Träff and Andreas Ripke. An optimal broadcast algorithm adapted to SMP-clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 12th European PVM/MPI Users' Group Meeting*, volume 3666 of *Lecture Notes in Computer Science*, pages 48–56, 2005.

4. Rolf Rabenseifner and Jesper Larsson Träff. More efficient reduction algorithms for message-passing parallel systems. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 36–46, 2004.

5. Jesper Larsson Träff. An improved algorithm for (non-commutative) reduce-scatter with an application. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 12th European PVM/MPI Users' Group Meeting*, volume 3666 of *Lecture Notes in Computer Science*, pages 130–138, 2005.

6. Peter Sanders, Jochen Speck, and Jesper Larsson Träff. Full bandwidth broadcast, reduction and scan with only two trees. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 14th European PVM/MPI Users' Group Meeting*, volume 4757 of *Lecture Notes in Computer Science*, pages 17–26. Springer, 2007.

7. Peter Sanders and Jesper Larsson Träff. Parallel prefix (scan) algorithms for MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI Users' Group Meeting*, volume 4192 of *Lecture Notes in Computer Science*, pages 49–75. Springer, 2006.

8. Jesper Larsson Träff. Hierarchical gather/scatter algorithms with graceful degradation. In *International Parallel and Distributed Processing Symposium (IPDPS 2004)*, page 80, 2004.

9. Jesper Larsson Träff. Efficient allgather for regular SMP-clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI Users' Group Meeting*, volume 4192 of *Lecture Notes in Computer Science*, pages 58–65. Springer, 2006.

10. Jesper Larsson Träff. Improved MPI all-to-all communication on a Giganet SMP cluster. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 9th European PVM/MPI Users' Group Meeting*, volume 2474 of *Lecture Notes in Computer Science*, pages 392–400, 2002.

11. Peter Sanders and Jesper Larsson Träff. The hierarchical factor algorithm for all-to-all communication. In *Euro-Par 2002 Parallel Processing*, volume 2400 of *Lecture Notes in Computer Science*, pages 799–803, 2002.

The general framework for implementation of the collectives in MPI/SX is described in the overview paper [2] of Chapter 6. The MPI/SX collectives are in general fully aware of the SMP structure of the SX system, and in most cases efficiently adopted to the communication restrictions of such systems. The algorithms developed are hierarchical in various ways and exploit special point-to-point functions of the MPI/SX ADI for communication between nodes, and mostly direct shared-memory communication between processes on the same SMP node. In the software architecture a distinction is made at communicator creation time between three cases: single-node, one process per node, and general multi-node case. The case where a communicator includes only one MPI process per SMP node is called the *flat* case, and is considered important because of applications programmed in a hybrid OpenMP+MPI paradigm that typically have only a single, active MPI process on the SMP nodes. For each of the three cases a separate function table is created which allow collectives

to immediately call the relevant implementation. Also, data structures in the communicator are set up to permit constant time computation of the number of processes per node, and node relative, local (consecutive) ranks of processes. In addition to saving latency at run-time, this case analysis has the software engineering advantage of separating the code for cases that are largely orthogonal. Some code duplication, of course, is inevitable. Both intra- and inter-node communication is done via non-swappable global memory segments. To save latency, small global memory buffers have been preallocated for each process. To save memory, larger global memory buffers are allocated on demand and freed again after use. There is a maximum, parametrizable size of such buffers, and all collective algorithms employ either blocking or pipelining to stay within this maximum buffer size. If allocation fails at a process, this process falls back to an implementation using only local memory buffers. This is not too difficult since the communication functions of the ADI works on both global and local memory buffers. Thus, no explicit synchronization and agreement on intermediate buffer size or algorithm/implementation is necessary for the collectives.

In addition to the ADI send and receive functions described in Chapter 8 yet another MPI/SX ADI function MPIR_Recv_func (and MPIR_Send_func for MPI/EX) for overlapping node internal buffer copying with inter-node communication was introduced. This receive function takes a function and parameter pointer as additional arguments, and schedules the function call for evaluation after the asynchronous IXS transfer has been set up. IXS data transfer and function evaluation can, for longer transfers to a global memory segment, proceed concurrently.

The reservoir of collective algorithms available in MPI/SX is quite large, and the different algorithms sometimes have merits under different circumstances, e.g. depending on the exact communication capabilities of the system (SX IXS and Infiniband/Myrinet switches), on the application load, and so on. To some extent MPI/SX chooses among alternatives based on environmental conditions, but there has been no attempt at incorporating a general framework for *automatic* selection among a large number of essentially *black box* algorithms, as suggested and advocated in for instance [FVD00, PGFA+07]. One obvious reason is of course that the algorithms available for MPI/SX are not in a black box, but have (in principle) well understood, parametrizable properties. Therefore much more deliberate, and earlier (e.g. at communicator creation time) choices can be made.

On the other hand, the collective MPI/SX algorithms are often parametrizable and designed with the goal of covering a large spectrum of data and communicator sizes with just one single algorithm. To get best performance a lot of situation dependent parameter tuning is necessary, and it would be a challenging task to automate some of this.

The papers are commented on below, and contain references to the relevant literature. There has been a very large amount of work on algorithms and optimizations for collectives in recent years, and no systematic literature survey shall be attempted here. A few immediately relevant papers, also in taking up some of the work documented here are [TNP03, TGR04, BGP+94, CHPvdG07, PGAB+07].

The conference contributions [2], [3] and [6] have meanwhile (in 2008) appeared or been accepted to appear (in 2009) in journal form. These papers should thus be considered

superseded by:

- Jesper Larsson Träff and Andreas Ripke. Optimal broadcast for fully connected processor-node networks. *Journal of Parallel and Distributed Computing*, 68(7):887–901, 2008. Supersedes [2] and [3].

- Peter Sanders, Jochen Speck, and Jesper Larsson Träff. Two-Tree Algorithms for Full Bandwidth Broadcast, Reduction and Scan. *Parallel Computing*, to appear 2009. Supersedes [6].

9.1 Algorithmic improvements

The algorithmic improvements achieved for the MPI collectives are summarized in Table 9.1. These are explained in some detail in the following sections and in the actual papers in Part III. As usual p denotes the number of processes in the calling communicator, and logarithms are always to base two. The problem size is denoted by m. The precise meaning should be clear from context.

The collective algorithms are mostly designed for systems with single-ported, bidirectional (send-receive) communication. Communication between SMP nodes is assumed to be fully connected and homogeneous. The design and analysis of the collective algorithms is almost exclusively done in the linear communication cost model. For design and evaluation of some of the collectives in the *LogP* model, see for instance [BIC05, Ian97, San02]. It is likely that k-ported collectives will become more important in the coming years. Small, first steps towards developing k-ported MPI collectives has been taken in [CvdGGT06, QA06, QA07a, QA07b].

9.2 Barrier synchronization

Barrier synchronization in MPI/SX for communicators spanning multiple nodes first and foremost has to exploit the dedicated IXS hardware barrier counters. The SX hardware barrier counter feature consists of two registers, one of which can be preset and decremented atomically, the other one being a toggle register which alternates between zero and one as the counter register reaches zero. This functionality obviously makes implementation of MPI_Barrier for any multi-node communicator straightforward. For performance reasons, MPI_Barrier is implemented hierarchically, with a node internal barrier implemented using atomic (test-and-set) operations on node local communication registers.

The problem is of course that hardware barrier counters are a scarce resource, and that only a few can be allocated per MPI/SX job. The scarce resource is managed within MPI/SX, which maintains the following, simple resource invariant: each barrier counter is either assigned to a communicator, or kept by exactly one MPI process. When a new communicator is created, the barrier counter of the process with the smallest rank that has a counter is, if any such process exists, assigned to the communicator. When the communicator is freed, the barrier counter is (arbitrarily) kept by the process with rank zero. This simple

MPI_Bcast	Optimal number of rounds $N - 1 + \lceil \log p \rceil$ for N blocks. Optimal running time $(\sqrt{(\lceil \log p \rceil - 1)\alpha} + \sqrt{\beta m})^2$ in linear cost model. Two tree algorithm.
MPI_Reduce, MPI_Allreduce	Improved butterfly algorithm for non-power-of-two case. Two tree algorithm.
MPI_Reduce_scatter	Improved butterfly algorithm for non-power-of-two case. Butterfly algorithm for non-commutative, associative operators. Adaptive gathering for irregular problems.
MPI_Scan	Doubly pipelined scan. Two tree algorithm.
MPI_Gather, MPI_Scatter	Binomial tree with graceful degradation to linear algorithm.
MPI_Gatherv, MPI_Scatterv	Adaptive tree construction Graceful degradation from adaptive tree to linear algorithm.
MPI_Allgather	Simultaneous binomial trees with graceful degradation to linear ring.
MPI_Alltoall	Hierarchical factor algorithm for irregular SMP nodes.

Table 9.1: Summary of algorithmic improvements for MPI collectives.

scheme requires no extra communication, but it can easily happen that a new communicator does not get a barrier counter even if other (albeit unrelated) processes had barrier counters. Some limited balancing of resources is done. When a new communication domain is spawned dynamically, or when communicators are merged, available barrier counters are exchanged between the root processors is the operation.

For communicators not getting a barrier counter, a software barrier implementation is used as fallback. Since this is decided at communicator creation time, a function table entry can be set accordingly, causing no decision overhead in the actual MPI_Barrier calls. Software barriers are implemented hierarchically with a simultaneous binomial tree scheme [BHK+97] for the barrier across nodes, and shared memory synchronization trees inside the SMP nodes [MCS91].

9.3 Broadcast

Broadcast is one of most deeply studied collective communication problems, and there is a wealth of theoretical results in various communication models, from which also practical implementations for MPI can profit. Some older, classical overviews are can be found in [FL94, HHL88, DSU04]. Broadcast algorithms can often, but not always, be used for the design of reduction and allgather algorithms. The MPI_Bcast collective is a broadcast operation for fixed sets of processes with known root and flexible rules for the structure of

the data to be broadcast.

MPI implementations typically use either (or a combination) of the binomial trees, skewed or Fibonacci trees, (pipelined) binary trees, scatter/allgather, and/or linear pipeline algorithms for the implementation. Apart from skewed trees, which can give theoretically better performance for some values of $LogP$ parameters, all these algorithms have been implemented within MPI/SX and benchmarked against each other.

The scatter/allgather broadcast algorithm which may be less well-known than some of the other algorithms mentioned probably goes back to the ideas in [BGP+94] and is used as basis for MPI_Bcast in for instance `mpich2` [TGR04]. To broadcast m data the data are divided into $m/(p-1)$ roughly even sized blocks that are scattered over the $p-1$ non-root processes. An allgather operation over the $p-1$ non-root processes completes the broadcast.

The first paper [1] gives an algorithm and implementation of MPI_Bcast reminiscent of the scatter/allgather idea, although it is not a direct, naive implementation. The algorithm is primarily intended for the flat, homogeneous case with only one MPI process per SMP node. The m data are scattered along a binomial tree. In each communication round each process halves the amount of data that it sends further down the binomial tree. The binomial tree can be sliced into levels, level k for $k \geq 0$ consisting of the processes that receives data in round k. The amount of data received is $m/2^k$, and in subsequent rounds each such process sends $m/2^k$ data further down the tree. At each level, all m data are present, so instead of a global allgather operation, $\log p$ level-wise, independent allgather operations are done concurrently to complete the broadcast. The allgather step is done by a butterfly algorithm (see Section 9.4) and takes k steps. The advantage of this algorithm is that it is possible to stop the halving process after any round k and continue as in a straightforward binomial tree algorithm for all following rounds. This can prevent the $m/2^k$ sized blocks from becoming too small. The algorithm [1] thus has the binomial tree algorithm as a genuine special case, and can therefore be used for the whole spectrum of problem sizes m. The *halving threshold* at which the switch to binomial tree algorithm takes place is a tunable, system dependent parameter. Another, nice property of the algorithm is that each block of data is received exactly once by each of the non-root processes. As will be explained, the algorithm is less well-suited for generalization to the multi-node SMP case with more than one process per node.

The algorithm can be formulated very nicely when the number of processes p is a power of two. When this is not the case, the solution is less elegant. Let $p = 2^{k_0} + 2^{k_1} + 2^{k_2} + \ldots$, where all k_i are distinct (in which case the decomposition is unique). The processes are grouped into subsets of size 2^{k_i}. The scatter step of the broadcast algorithm is executed as described for the largest subset (which must contain the root process) of 2^{k_0} processes. Before the level-wise allgather step at level k_i starts, the processes at this level send their data of size $m/2^{k_i}$ to the processes in the group of size 2^{k_i}. After this, all levels can allgather concurrently as described above.

In the best implementation, as for instance achieved in [1], the number of communication rounds is $\lceil \log p \rceil$ for both scatter and allgather step, and each takes time proportional $\alpha \lceil \log p \rceil + \beta m$ in the linear communication cost model. The total time is therefore

$2(\alpha\lceil\log p\rceil + \beta m)$, and comparable to the time taken by a pipelined binary tree algorithm. This is theoretically about a factor two from optimal, and was the main motivation behind the further work on broadcast algorithms.

It is well-known that in the homogeneous, single-ported, linear cost model, the lower bound for broadcasting m data to $p-1$ processors is $\max\{\alpha\lceil\log p\rceil, \beta m\}$. The first term is due to the fact that by the single-ported restriction the number of processors that have received data can at most double in each communication round. The second term simply states that the m data eventually have to leave the root process. With the same arguments it can also be shown that if the m data are divided into N blocks, $N-1+\lceil\log p\rceil$ communication rounds are required to broadcast the N blocks.

It was an open problem for a long time during the 1990ties to achieve this lower bound for arbitrary number of processes (the influential paper [JH89] showed the result for hypercubes). Not known to the author before completion of the work in the papers [2] and [3], this was achieved for the single-ported, bidirectional, send-receive model in [BNKS00, KC95]. Both of these papers, however, only sketch a theoretical solution, and none of them present any implementation results.

A different, original solution to the theoretical problem of reaching the lower bound on broadcast in the single-ported, send-receive model was first presented in [2]. The solution uses an approximate binomial tree for distribution of the first $\lceil\log p\rceil$ blocks of data, and the same communication pattern is reused in the remaining $N-1$ rounds. Thus, the algorithm has the advantage of having a(n approximately) binomial tree as a special case, and can therefore be used also for small data. In other words, the same algorithm can span the whole range of problem sizes m, and works for arbitrary number of processes p. For MPI implementations such properties mean that only one and the same algorithm has to be implemented, debugged, tuned and maintained. The solution to the broadcast problem in [2] relies on the construction of a schedule which determines for each process and each round, which block of data is to be received, and which (already received) block is to be passed further on. The schedule repeats after $\lceil\log p\rceil$ rounds, thus only $O(\log p)$ space is required per process for storing the schedule. A drawback, and an unsolved problem of the approach is that the schedule construction is costly. An off-line algorithm taking $O(p\log p)$ steps is given in [2]. This is, however, good enough to be useful for MPI libraries, since the schedule can be constructed at communicator creation time and the construction time amortized over a number of subsequent broadcast operations. Still, it is an interesting question, whether it is possible to give a solution where the schedule for each process can be computed in $O(\log p)$ steps only. This was recently answered in the affirmative in [Jia07] by using a different communication pattern.

Dividing the problem into N roughly equal-size blocks of size m/N the broadcast time in the linear cost model is $(N-1+\lceil\log p\rceil)(\alpha+\beta(m/N))$. An optimal block size can be found by simple calculus (or by simply balancing the terms αN and $(\lceil\log p\rceil-1)\beta(m/N)$) to find the optimal broadcast time. See Table 9.1.

Because of its communication pattern the optimal broadcast algorithm [2] is termed the *circulant graph algorithm* in a more the more comprehensive journal version of the papers [2]

and [3]. In the paper [2] the description of the algorithm is divided into two cases, depending on whether $N-1$ is a multiple of $\lceil \log p \rceil$. The solution when this is not the case is quite clumsily described, and the case can in fact be handled very elegantly and easily. All that has to be done is to add a number x of *dummy* communication rounds *before* the broadcast proper starts, so that $x+N-1$ is indeed a multiplum of $\lceil \log p \rceil$. In the dummy rounds, no data are sent and no data received. This is described more clearly and in detail in the version journal of the papers.

Pipelined algorithms are in general well-suited to implementation on SMP systems (there is no claim made that this is in a theoretical sense optimal). The idea is simple. The algorithm is run over a set of representative processes, one (local root) for each SMP node. When a new block is received it can be distributed (or otherwise processed) among the remaining processes on the SMP node, simultaneously with the algorithmic actions of the representative process. For the optimal broadcast algorithm this is described in the paper [3] (with some repetition of the algorithm description of [2]). In both papers [2] and [3] performance results on a small AMD Athlon cluster with Myrinet interconnect and comparisons to some of the other, above mentioned broadcast algorithms are given. An improvement of a factor 1.5 over pipelined binary trees and even more for scatter/allgather is achieved. The algorithm is also implemented in MPI/SX and give similarly good results on more than 30 SX-nodes, both in the flat and in the general multi-node case. In the paper [6] the algorithm is benchmarked against other algorithms on 150 nodes of a dual-core Intel Xeon cluster with Infiniband interconnect. Improvements consistent with the other systems are achieved. In particular, the algorithm is even for extremely large problems on par with the simple, linear pipeline (which is also bandwidth optimal for $m \gg p$) that is completely unusable for small problems.

In the paper [6] a completely new approach to broadcast, reduction and parallel prefix was proposed, which sets out to analyze and alleviate the drawbacks of the binary tree algorithm. The drawbacks of the binary tree are that interior nodes receive a new block only in every second communication round, and thus only exploit the bidirectional communication capabilities half of the time, and that leaves (of which there are $p/2$) only receive blocks, and thus never exploit the bidirectional communication capabilities.

The solution is to use two binary trees. These are constructed such that the leaves of the one tree are interior nodes of the other and vice versa. Thus, in any two successive communication rounds, each process is receiving a block in its role as leaf and another block in its role as interior node, and is in each round sending a block to one of its children. To ensure that the nodes are in each round assigned either the role of interior or leaf node, but not both, and thus to ensure that each process is receiving and sending a block in each round, an ordering of the incoming and outgoing edges of the nodes is required. A simple construction shows that this can be achieved by edge two-coloring of a bipartite graph with maximum degree two. The communication schedule for the *two tree algorithm* can thus be computed in "only" $O(p)$ steps, slightly better than the schedule for the circulant graph algorithm. The idea of using two binary trees was also proposed in [HV05], but the need for scheduling between the two trees not realized.

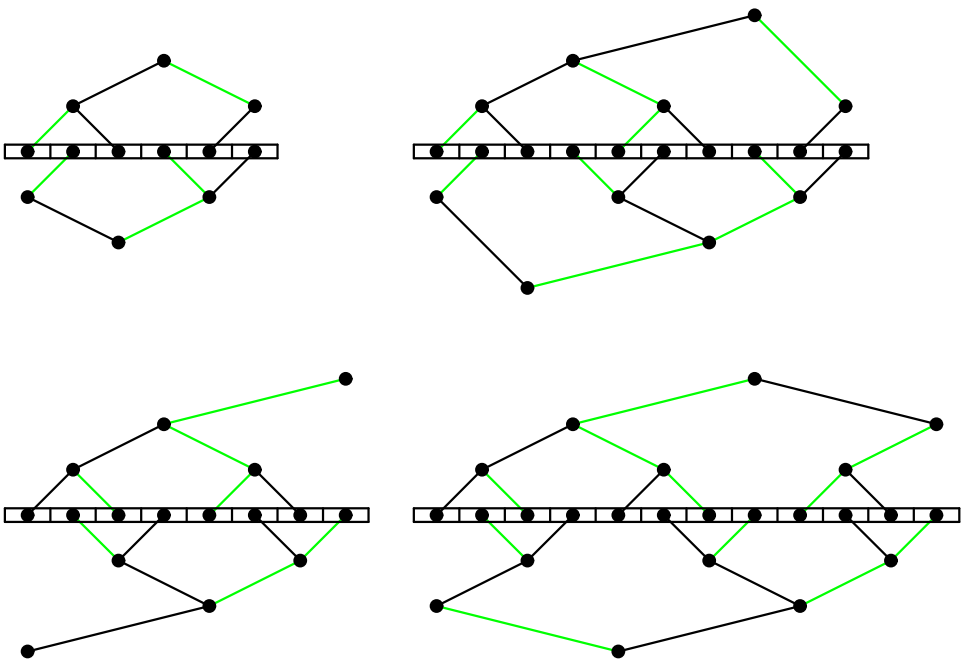

Figure 9.1: The construction of the two trees by *mirroring* for $(p-1) \in \{6, 8, 10, 12\}$. The lower tree T_2 is the mirror image along an imaginary diagonal of the upper tree T_1. The root process is external to the two trees and connected to each by an edge.

function inEdgeColor(p, i, h)
  if i is the root of T_1 **then return** 1
  while $(i \sqcap 2^h) \equiv 0$ **do** $h \leftarrow h + 1$ // compute height
  $i' \leftarrow \begin{cases} i - 2^h & \text{if } (2^{h+1} \sqcap i) \equiv 1 \vee i + 2^h > p \\ i + 2^h & \text{otherwise} \end{cases}$ // compute parent of i
return inEdgeColor$(p, i', h) \oplus [(p/2 \bmod 2) = 1] \oplus [i' > i]$

Figure 9.2: Pseudocode for coloring the incoming edge in T_1 of an even numbered rank i (assuming here that processes are ranked $1..p$). The parameter h is a lower bound on the height of node i in T_1. $[P]$ converts the predicate P to a value 0 or 1. $\sqcap$ denotes bitwise *and*, and $\oplus$ logical exclusive or.

In the paper [6] it is indicated how the edge coloring can be computed locally by each process in only $O(\log p)$ steps, thus eliminating the need for an off-line construction at communicator creation time. To achieve this, a different construction of the two trees than used in the paper is needed. The construction is called *mirroring*, and is completely symmetric as is needed for the proof. The mirror construction is shown in Figure 9.1 for values of $p = 6, 8, 10, 12$. With this construction of the two trees, the color of the incoming edge of the upper tree can be computed by the algorithm in Figure 9.2, and the colors of the other incoming edge as well as the two outgoing child edges follow easily from this. The proof of correctness of the construction is not given in [6], but can be found in the full journal version of the paper.

To use the two tree algorithm, half of the m data are broadcast over the first tree, and the other half of the data over the second tree. Since the two broadcasts take place concurrently, the total time is the same as the time for a simple, pipelined binary tree broadcast of $m/2$ data. In the linear time communication cost model the best that can be achieved is

$$2(\lceil \log p \rceil - 1)\alpha + 4\sqrt{(\lceil \log p \rceil - 1)\alpha}\sqrt{\beta m/2} + \beta m$$

This is comparable to the best possible time for the circulant graph algorithm, with a factor of two worse latency (the paper [6] makes a mistake in the statement of the time, multiplying the latency with two).

The two tree algorithm can also be used for reduction and parallel prefix, see Section 9.4.

9.4 Reduction and Parallel Prefix

Standard algorithms for the implementation of the MPI reduction collectives (MPI_Reduce, MPI_Allreduce, MPI_Reduce_scatter, MPI_Scan, MPI_Exscan) are based on binomial trees, skewed or Fibonacci trees, (pipelined) binary trees, butterflies, and linear pipelines. Variants of all of these algorithms, except for skewed trees (again, typically occuring in designs withing the *LogP* or postal model), have been implemented and benchmarked against each

other within the framework of MPI/SX. Some findings and new developments are described in the papers discussed below.

The tree based algorithms are similar to the corresponding algorithms for broadcast with the direction of the communication edges reversed, and it is often possible to convert a broadcast algorithm into an algorithm for rooted reduction in this way.

Due to the special requirements of MPI that reductions be performed in canonical (rank) order (and that binary reduction operators are not always commutative), and that the same reduction order and bracketing (evaluation order of subexpressions) be used for all elements of the input vectors, this is, however, not always possible. The optimal send-receive model broadcast, for instance, cannot be used for non-commutative operators, and it would also not be possible to guarantee the same bracketing. Also broadcast based on the scatter/allgather idea can only with care be used for reduction. The variant in [1] could, however, quite easily be transformed into a rooted reduction algorithm, since the allgather step is implemented by a butterfly algorithm.

Butterfly algorithms are well-suited for implementation of the MPI reduction collectives MPI_Reduce, MPI_Allreduce and MPI_Reduce_scatter, and are also theoretically attractive.

As in Section 3.5 let x_i denote the input vector for process i, $0 \leq i < p$, and first assume that p is a power of two. In a *butterfly algorithm* process i exchanges a partial result with process $i \oplus 2^k$, where $\oplus$ denotes *bitwise exclusive or* in round k for $k = 0, \log p - 1$, and computes a new partial result. It is easy to maintain that processor i has computed a partial result of the form $\bigotimes_{j=i\wedge\overline{(2^{k+1}-1)}}^{i\vee(2^{k+1}-1)} x_j$ where $\wedge$ denotes *bitwise and*, $\vee$ *bitwise or*, and $\overline{x}$ *bitwise, binary complement* of x. For long vectors $|x_i| = m$ an algorithm that is linear in m can be obtained by halving the size of the partial result vectors in each round, and collecting the result in another $\log p$ round gathering phase. This is well-known, see for instance [vdG94]. For reduction-to-all (MPI_Allreduce) the second phase is an allgather operation.

The butterfly algorithm is strictly only for power of two number of processes. Let p' be the largest power of two smaller than p. To exploit the butterfly algorithm for non-powers of two, a trivial solution is to first pair up the first $2(p - p')$ processes in $(i, i + 1)$ pairs, let the smaller numbered process in each pair receive data from the larger and perform a reduction step, and then run the butterfly algorithm on the p' processes formed by the smallest processor in each pair and the remaining non-paired processes.

The trivial solution increases the time by one communication round of m data. This is improved for large m for MPI_Reduce, MPI_Allreduce (and implicitly for MPI_Reduce_scatter) in the paper [4]. Instead of process $i + 1$ of the pair $(i, i + 1)$ sending its input to process i this first step is combined with the first round of the butterfly algorithm, in which a *triple exchange* step is done instead of the simple exchange step described above. In the triple step processes i and $i + 1$ first exchange half their data, and perform a reduction. In the next step process i receives from $i \oplus 2^k$ and the partner receives not from i but instead from $i + 1$. This reduces the extra communication volume when p is not a power of two to $m/2$, still adding only one extra communication round. This idea can be taken further, much as explained for the scatter/allgather broadcast algorithm [1]. Let $p = 2^{k_0} + 2^{k_1} + 2^{k_2} + \ldots$ with different k_i. Then the processes can be organized such that the butterfly algorithm can be

used in each group of size 2^{k_i} with triple steps with data of size $m/2^{k_i+1}$ performed after $k_i + 1$ communication rounds. This further reduces communication volume, depending on the decomposition of p into smaller powers of two. Two variations of this idea, also for the MPI_Allreduce collective are given in [4]. This paper gives no implementation results, but the improved butterfly algorithm is used as the basis for MPI_Reduce, MPI_Allreduce, and MPI_Reduce_scatter implementations in MPI/SX. As for the broadcast algorithm [1] halving can be stopped when $m/2^{k+1}$ becomes too small. This saves the same number of allgather steps in the second phase of the algorithms. For the built-in commutative MPI operators, these algorithms are adopted to the SMP case with more than one process per node by preceeding the butterfly algorithm with a node local reduction.

The improved butterfly algorithm is exploited for the implementation of MPI_Reduce_scatter in the paper [5]. In this collective the result vector of size $m = \sum_{i=0}^{p-1} m_i$ will be scattered in blocks of size m_i over all processes, and it might therefore be possible to save the allgather phase of the butterfly algorithm. This was claimed not to be possible in [Ian97, BIL02, TG03], the latter therefore using the butterfly algorithm only for commutative operators.

The problem with the butterfly algorithm is only that the result blocks end up in permuted order at the processes. Block m_i will end at process $\sigma(i)$ for a permutation σ determined by the butterfly communication pattern. The simple solution given in [5] is to permute the input blocks locally on each process *before* the butterfly algorithm, that is to put block $m_{\sigma(i)}$ in position i.

A final problem is caused by the fact that MPI_Reduce_scatter collective is *irregular*. Supplying blocks of different sizes m_i would compromise the complexity of the algorithm by a logarithmic factor (for the extreme case where one block has size m and all others size 0). This problem is also solved in the paper [5]. The butterfly part of the algorithm is done on regular, roughly equally sized blocks, and an adaptive reordering step of at most $\lceil \log p \rceil$ rounds is added.

The MPI standard has regular and irregular variants for the data exchange collectives. Having only an irregular MPI_Reduce_scatter operation is not consistent with the spirit of the MPI standard. There are situations where a regular variant of MPI_Reduce_scatter is all that is needed, for instance in the implementation of MPI_Win_fence as described in Chapter 8, but see also [GRM04]. Finally, an implementation of a regular variant can be faster by not having to cater for the adaptive reordering. A regular variant is proposed by the author to the MPI Forum with the name MPI_Reduce_scatter_block suggested by William Gropp. This collective is likely to be included in the MPI 2.2 version of the MPI standard.

Like for the other reduction collectives there are a number of folklore algorithms that can be used for implementation of the scan or parallel prefix collectives. Examples are (pipelined) binary trees, binomial trees, simultaneous binomial trees, and linear pipelines. Many of these algorithms have been implemented in the framework of MPI/PC and comparison results and some minor improvements were presented in the paper [7].

The pipelined binary tree algorithm, described for instance in [MP93], consists of an up-phase very similar to reduction and a down-phase very similar to a broadcast. An elaborate

implementation of this algorithm in which the two phases are overlapped is given in the paper [7], and compared to some of the other algorithms mentioned above. A worthwhile (but expected) confirmation is that the linear pipeline for large vectors performs substantially better than any of the other algorithms. It might therefore be recommendable to implement MPI_Scan by a hybrid algorithm with a (simultaneous) binomial tree algorithm for small, and a linear pipeline for large problems. This is followed in MPI/SX.

The two tree idea of the paper [6] described for broadcast can be used for both reduction and parallel prefix. For both they give, presumably, the theoretically currently best known results of the form $O(\log p + m)$. The idea can only be used for a restricted reduction problem in which the root is either process 0 or process $p-1$. A somewhat serious limitation of the two tree idea for MPI reduction operations is that since the two trees are structurally different the two halves of the input data will not be reduced with the same bracketing.

9.5 Gather and scatter

The regular gather-scatter operations of MPI are usually implemented by either (or a combination of) a straightforward, linear algorithm in which all non-root processes send data to or receive data from the root process, or by simple binomial (or skewed) trees. In case a combination of these algorithm are used, finding the best switch point which gives the smoothest performance might be difficult. A pure tree-based algorithm which require intermediate buffering for the non-root processes can be too space consuming for large systems.

The paper [8] proposes a binomial tree with *graceful degradation* towards a simple linear algorithm as problem size increase. Degradation will start from subtrees as intermediate buffer size reach a predefined threshold. This gives a smooth transition from the fast binomial tree algorithm for small problems towards the space and bandwidth efficient, linear algorithm for large problems. It is also shown how this algorithm can be used for SMP systems, simply by using a consecutive, virtual ranking of the processes for the construction of the tree. The processes are semi-sorted according to the index of the SMP node to which they belong, and the processes are ranked in that sorted order. This information is exactly what is computed at communicator creation time and stored with the communicator, so there is no extra cost implied by the virtual reranking step. The resulting tree will have the property that only few tree edges cross SMP nodes, and that communication on such edges tend to take place in different rounds. There is no proof of optimality of the construction in the paper [8], and optimality, if at all obtained, would depend on the relative intra- and inter-node communication performance. For random communicators, the construction is clearly better than constructing the binomial tree in communicator rank order.

The algorithms for the irregular MPI_Gatherv and MPI_Scatterv collectives use a simple weight-doubling algorithm to construct trees dependent on the data sizes to be gathered from or scattered to the non-root processes. All data sizes smaller than an intermediate buffer threshold are sorted in decreasing order. The ith subtree T_i is constructed recursively from the smallest segment of processes in this sequence such that the sum of the sizes of nodes in T_i is larger than or equal to the sum of all sizes before the start of segment T_i. This

results in binomial like trees with small idle times. Again, no optimality result is proved.

For MPI and the practical implementation, a further difficulty is that only the root process has the full information to construct the tree. No other process can find its position (parent and children) in the tree without guidance from the root or from other processes. An algorithm which allows a partly distributed construction of the tree by sending segments of the sorted size list to roots of subtrees is developed in the paper [8]. This is the first such algorithm for an MPI library. The resulting MPI_Gatherv and MPI_Scatterv implementations performs considerably better than the simple, linear algorithm for small problems, and degrades gracefully to the performance of the linear algorithm as problem size increases. The overhead of the distributed tree construction seems small enough to make such a construction worthwhile.

9.6 Broadcast-to-all

For the flat, one process per SMP node case an often used algorithm for implementation of MPI_Allgather is the simultaneous binomial tree algorithm given in [BHK+97] (under the name *catenation*). This algorithm is not suitable for SMP systems, as shown both in the paper [3] of Chapter 10 and in the paper [9]. In some of the $\lceil \log p \rceil$ communication rounds it will inevitably happen that more than one process per SMP node will have to send or receive data from another node, leading to serialization and performance degradation in single-ported systems. For regular SMP nodes with the same number of processes per node the algorithm can be extended by a simple hierarchical approach. First gather locally on each SMP node, then run the algorithm on the set of nodes, and complete the allgather operation by a local broadcast. This in general implies allocation of an intermediate buffer proportional to the total size of the allgather problem, which is untenable for large problems. For SMP systems a linear ring, with a virtual ranking defining the predecessor and the successor easily achieves that in each of the $p-1$ communication rounds exactly one process per node receives data from another node and exactly one process per node sends data to another node. To implement MPI_Allgather these two algorithms have to be combined, which typically leads to large performance jumps at the switch point.

An idea for more graceful degradation from the catenation algorithm to the linear algorithm is given in the paper [9]. The idea is simply to run the catenation algorithm until the intermediate buffer is exhausted, at that point switching to the linear ring algorithm. When this happens several, disjoint rings will be active concurrently. In the MPI/SX implementation, the local gathering and broadcasting of data is furthermore done in a pipelined fashion, simultaneously with the sending and receiving of data. For this the MPI/SX ADI MPIR_Recv_func function is useful. The benchmark results show a much smoother transition between the two algorithms, an effect that will be amplified with increasing number of processes.

9.7 Personalized all-to-all

A trivial algorithm for solving the regular, personalized all-to-all communication problem MPI_Alltoall in MPI in the bidirectional send-receive model performs p send-receive rounds. In round k for $k = 0, \ldots, p-1$ process i sends a block to process $(i+k) \bmod p$ and receives a block from process $(i-k) \bmod p$. Another well-known algorithm, which relies on only simple, telephone like bidirectional communication, is based on *factoring* or decomposing the complete communication graph into a sequence of p 1-factors (matchings; graphs in which each node has degree one). For each communication round each process exchanges a data block with its partner process. Both of these algorithms are of course not suited to SMP clusters, since in that case it can easily happen that several processes will at the same time be involved in communication with a process on another node, and thus cause serialization at the node communication port. The factor algorithm can be adapted to SMP systems by factoring over the nodes, though. The 1-factor algorithm is used in MPI/SX for the flat communicator case for longer messages. For small messages the logarithmic round, combining algorithm of [BHK+97] is used. A trivial algorithm for MPI simply posts p non-blocking receives and p non-blocking sends and wait for these communication operations to complete. This algorithm completely gives up on scheduling the communication and scenarios with both severe congestion and severe starvation can be constructed.

The papers [10] and [11] extend the factoring idea to SMP systems, in particular for the case where the communicator may contain a different number of processes on each node. The challenge is to decompose the exchange steps to avoid load imbalance by partner nodes with many processes starving partner nodes with few processes. The new algorithm runs in a number of phases, in each one going through a sequence of 1-factors, and getting rid of the nodes with the minimum number of processes among the nodes of the phase. This leads to an algorithm, called the *hierarchical factor* algorithm that is optimal up to an additive term if the number of nodes is large relative to the number of processes per node. For the case where each of the N SMP nodes has the same number of processes n, the hierarchical factor algorithm is identical to a factoring algorithm over the nodes, and takes one phase of N exchange rounds.

In [11] the hierarchical factor algorithm is implemented and benchmarked on a small Giganet cluster, and compared to other algorithms for MPI_Alltoall. Small but worthwhile improvements are shown on this (small) system. It is shown that the individual communication steps can be scheduled such that at most one process at a time will be using the communication port. The practical effect, investigated on the Giganet cluster, turned out not to be significant for this system.

The hierarchical factor algorithm is also used in MPI/ES, but benchmark results are not available. The benefits of factoring are huge compared to the trivial MPI algorithm with unscheduled, non-blocking send and receive operations. For small MPI_Alltoall problems messages are grouped in the exchange step to save on latency.

9.8 Irregular data exchange collectives

The papers [8] and [5] present (slightly) nontrivial, improved algorithms for the irregular MPI collectives MPI_Gatherv, MPI_Scatterv, and MPI_Reduce_scatter, but the irregular collectives remain a major challenge – both theoretically and for practical implementations. For the latter, the fact that the information about the optimization problem to be solved is only available in a distributed fashion, so that information first has to be gathered if a centralized algorithm is to be used, already implies overhead that may easily render a theoretically good solution uninteresting for small problems. For practical implementations means to hide the gathering overhead therefore have to be found, as was sketched for the MPI_Gatherv and MPI_Scatterv collectives in the paper [8]. Theoretically defining the optimization problem and relevant machine model is a first step, and in many cases the optimization problem will turn out to be NP-complete. The best that can be hoped for are thus fast, possibly distributed heuristics with good practical performance. For the important MPI_Alltoallv operation frameworks and some heuristic solutions have been considered in [RWK95, AG06, LWP02].

Chapter 10

Process topologies

As one of few MPI implementations MPI/SX has a non-trivial implementation of the MPI process topology functionality for systems with a hierarchical, SMP-like communication structure. For both the MPI graph and Cartesian topologies the MPI/SX implementation approximately solves a graph partitioning problem, which is a suitable and useful formulation of the remapping problem for such systems. A description of the MPI/SX implementation as well as a critical analysis of the MPI process topology functionality itself is contained in the following four publications.

1. Jesper Larsson Träff. Implementing the MPI process topology mechanism. In *Supercomputing*, 2002. `http://www.sc-2002.org/paperpdfs/pap.pap122.pdf`.

2. Jesper Larsson Träff. Direct graph k-partitioning with a Kernighan-Lin like heuristic. *Operations Research Letters*, 34(6):621–629, 2006.

3. Jesper Larsson Träff. SMP-aware message passing programming. In *Eigth International Workshop on High-level Parallel Programming Models and Supportive Environments (HIPS03), International Parallel and Distributed Processing Symposium (IPDPS 2003)*, pages 56–65, 2003.

4. Guntram Berti and Jesper Larsson Träff. What MPI could (and cannot) do for mesh-partitioning on non-homogeneous networks. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI Users' Group Meeting*, volume 4192 of *Lecture Notes in Computer Science*, pages 293–302. Springer, 2006.

The first paper [1] shows that the topology mapping problem for SMP systems can be solved by graph partitioning. The input to both MPI_Graph_create and MPI_Cart_create is, explicitly and implicitly, an unweighted communication graph. Assume that k SMP-nodes are spanned by the calling communicator. The number of MPI processes on each of the spanned nodes determines the sizes of k subsets of the p processes, and it is (perhaps) reasonable to assume that communication costs are minimized by minimizing the value of a

cut. In [1] various cuts are discussed, minimizing for example either the total number of cut edges, or the maximum number of cut edges between any two node subsets. Which is the best choice might depend on the application, but the MPI functionality does not provide means to convey such information to the implementation. It is pointed out in [1] that the restriction to unweighted graphs is a severe limitation. Weights could have been a means of transmitting information on the amount of communication between processes to the MPI library. Nevertheless, the MPI topology functionality can help in localizing communication and lead to application improvements. This is demonstrated by the benchmarks developed in [1]. In extreme cases, where a reordering exists that causes all communication to take place inside SMP nodes, very significant improvements can be achieved. Experiments were carried out on a 6-node SX-6 system.

A forerunner of the MPI/SX implementation was an implementation in the context of HP MPI in [Hat98], which was also based on graph partitioning. Apparently also IBM MPI has, or at some point had a non-trivial implementation of the process topology functionality [YCM06].

The relevant graph partitioning problem to be solved is partitioning into k subsets with prescribed sizes. This can be done by recursive bi-partitioning, which is a well-known NP-complete optimization problem [GJ79]. Heuristics are known that give very acceptable results in practice, fast, for instance the Kernighan-Lin heuristic [KL70].

However, bi-partitioning is more costly, by a logarithmic factor in the worst case, can give arbitrarily worse results [ST97], and makes it difficult to control the overall cut value. In the context of the MPI topology functionality it is therefore a relevant question whether it is possible to do better. The second paper [2] describes a generalization of the socalled Fiduccia-Mattheyses linear time variant [FM82] of the Kernighan-Lin heuristic to k partitioning with the *same* time bounds, that is, independent of k. This heuristic is used in MPI/SX, but the result is also of independent interest.

A critique of the MPI standard for process topologies is given in the third paper [3]. As mentioned above, the interface does not provide the possibility to influence the optimization criterion used for the reordering. Other points are

- vagueness and imprecision in the interface definition,
- lack of expressivity in the mechanism itself by relying on simple, unweighted graphs only, and
- lack of scalability/difficulty of use

The latter point relates specifically to the graph topologies, where each process has to provide the full communication graph. This consumes $O(n + e)$ space per process, n and e being the number of nodes and edges in the topology, respectively (with $n \leq p$ and $e \leq p^2$), and $O(p(n + e))$ space in total, and can become problematic for large systems. At least as important is that each process must supply the complete communication information for all the processes, even for applications where each process is expected to communicate with only a few neighbors. This is tedious for the user, and should better have been left to the MPI

library. Consider as analogy the irregular MPI_Alltoallv collective. Here each process only has to specify the amount of data to be sent and received from each of the other processes, and not what the other processes exchange among themselves. Even this can be bad enough in applications (in the sense that this information may not be readily available, and other collective operations may have to be invoked to collect it). In applications using one-sided communication, where the communication pattern may be particularly irregular and the user intention is not to bother about the exact pattern (otherwise, the point-to-point model could have been used), the requirement that each process must specify the full graph to MPI_Graph_create could be particularly burdensome, and prevent the use of the mechanism altogether. This problem is being addressed for MPI 2.2 which may well contain new graph constructors along the lines suggested in [3].

The critique in [3] is coupled with some proposals for MPI support functionality for more flexible translation of ranks and redistribution of data among calling and reordered communicators. There is also a proposal for more comparison result values for the MPI_Comm_compare operation.

The problem of allowing only unweighted graphs is addressed further in the last paper [4]. It is shown that for systems with a hierarchical communication system, like SMP clusters, the MPI topology mechanism is not sufficient for solving the load balancing problem arising in *mesh partitioning* applications. Here a large mesh/graph has to be partitioned into p subsets to minimize the communication between the p processes. Examples show that mapping this in a second step onto the actual SMP system with the existing MPI topology mechanism with unweighted communication graphs may give arbitrary bad results. And even if edge weights were allowed, the result could still be suboptimal. The fundamental problem is that the actual optimization problem to be solved consists in partitioning *and* mapping at the same time, which is a *quadratic assignment* problem, and not a graph partitioning problem. For a more accurate solution of the mesh partitioning problem on SMP systems, knowledge of the host communication system is needed. The paper discusses how to provide such knowledge to the application mesh partitioner in an abstract, portable way by adding *topology inspection functions* to MPI.

Chapter 11

Collective correctness and consistent performance

MPI operations with collective semantics have strict and sometimes intricate consistency requirements. It is easy for the user to make mistakes, and such may not have an immediate (bad) effect but cause behavior that is difficult to understand at a much later time. As an aid towards finding such mistakes in the use of collective functions a *verification library* has been developed for MPI/SX.

The lack of performance guarantees for MPI can sometimes tempt users to make optimizations that are valid only for a specific system with a specific MPI implementation, thus wasting time and hampering the performance portability of the application. This problem is addressed by discussing various self-consistent performance requirements that might be posed to MPI implementations.

These issues are dealt with in three conference contributions.

1. Jesper Larsson Träff and Joachim Worringen. Verifying collective MPI calls. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 18–27, 2004.

2. Jesper Larsson Träff and Joachim Worringen. The MPI/SX collectives verification library. In *Parallel Computing: Current & Future Issues of High-End Computing (ParCo 2005)*, volume 33 of *NIC Series*, pages 909–916. John von Neuman Institute for Computing, Central Institute for Applied Mathematics, Forschungszentrum Jülich, 2006.

3. Jesper Larsson Träff, William Gropp, and Rajeev Thakur. Self-consistent MPI performance requirements. *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 14th European PVM/MPI Users' Group Meeting*, volume 4757 of *Lecture Notes in Computer Science*, pages 36–45. Springer, 2007.

The conference contribution [3] has meanwhile (in 2009) been accepted to appear in journal form. The paper should thus be considered superseded by:

- Jesper Larsson Träff, William D. Gropp, and Rajeev Thakur. Self-consistent MPI Performance Guidelines. *IEEE Transactions on Parallel and Distributed Systems*, to appear 2009/2010.

As explained in Chapter 3, the MPI standard does not prescribe a particular behavior for erroneous (collective) MPI calls, and the behavior of different MPI implementations for the same erroneous call may differ. Even the same implementation may behave differently under different circumstances – or be entirely non-deterministic. A particular problem with erroneous (collective) calls is that the effects of an error may first show up long time after the call. At that point the reasons may be totally obscured.

The MPI collective operations have strict requirements on argument consistency, and not fulfilling these is a typical mistake in applications using collective operations. On the other hand, by these very requirements, and because all processes in the communicator will call the collective, collective operations are highly structured. This can be exploited in a natural way to use collective operations to check fulfillment of consistency requirements of other collective operations!

As required by Definition 1 (see Page 26), for each particular collective operation all processes in a communicator will eventually call the collective before calling any other collective. This condition will be termed *call consistency*. If not fulfilled, that is if in an application some processes are calling one collective and other processes another collective, the application will typically (but not always) hang. Call consistency can easily be validated by preceding (the implementation of) each collective operation by a broadcast from a selected process of information about which collective is being called by that process. Each other process can, upon entering its collective operation (where the broadcast is also performed), verify that it is indeed about to call the same collective as the selected process.

This idea can be used to verify all other consistency requirements. The MPI/SX verification library verifies that the same root argument is used in rooted collectives, that the same binary operator is used in reduction collectives (unless a user defined operator is used), that the `MPI_IN_PLACE` argument is used consistently (where allowed), that the same group is used in communicator creation collectives, that the same topology is specified, that information objects are used consistently, that size and flag arguments are identical for file operations, and so on.

A further, most important requirement is that datatype signatures between processes exchanging data in a collective operations must be identical. This is not strictly checked in the MPI/SX verification library. Being for the time being for systems with the same representation of basic datatypes, a necessary condition for the requirement to hold is that the lengths of the type signatures match. This is thoroughly checked. An exact check of type signatures would be possible by sending the compact representation of the type signature (already computed for one-sided communication) before the actual data communication. This would perhaps be defensible (in terms of incurred overhead) because $T \ll m$ for type DAGs with T nodes and type signatures of size m. Another approach is to compute a *signature* for the type (in another sense of the term) by hashing. An approach was first proposed in [Gro00] and later improved considerably in [LBFD05]. The hashing approach

to type signature checking is used in the MPI/SX inspired verification interface described in [FCLG05].

The error handling in the MPI/SX verification library is implemented to be symmetric. If a violation of a consistency requirement is detected at some processes, the error code is passed to all processes. All processes can then call the same error handler, which will normally lead to an abort (but could be set otherwise by the user).

A full summary of the checks performed by the MPI/SX verification library is given in Tables 11.1 and 11.2.

The original idea was to use the profiling interface facility of MPI [SOHL+98, Chapter 8] to implement the verification library. This would have made for a portable interface, but would have required duplicating a lot of work already done inside MPI/SX, and would have made certain checks more complicated. The approach for MPI/SX was to write a separate, thin library, with which the application must be linked if collective verification is to be performed. This has the advantage of keeping the MPI profiling interface vacant for other purposes together with the verification facility. The verification interface idea was quickly taken up by the Argonne National Laboratories, who used the MPI profiling interface for their implementation [FCLG05].

The first paper [1] introduces the idea of using collective operations to verify other collective functions, and presents the MPI/SX verification library. The second paper [2] measures the overhead entailed by collective verification, both with isolated benchmarks for collectives, and with an application which significantly uses collectives. For the latter a surprisingly small impact was found. The collectives benchmark shows the expected, significant overhead for small problems, sometimes up to a factor of 10. As problem size increases, the overhead caused by the additional, hidden collective calls can be amortized and tend not to be significant. Nevertheless, it is strongly recommended to use the verification library only during program development.

Dynamic verification of MPI applications has been addressed in other work both before and after the introduction of the MPI/SX verification library. The currently most active project is MARMOT [KBMR04, KMR04a, KMR04b, KMR06, KR06]. MARMOT is a larger project and checks both more and less than the MPI/SX verification library, including deadlock (other than call inconsistency), management of MPI objects, many locally checkable error conditions, and so on. Being implemented as a portable profiling interface it has to duplicate a lot of bookkeeping already done inside of the MPI library. Somewhat similar, but incomplete approaches were MPI Check (very incomplete, and only for Fortran) [LCC+03], and Umpire [VdS00]. The Intel Message Checker is partly more ambitious than the MPI/SX verification library by also checking aspects of point-to-point communication, but is much less complete and systematic on collectives, and, based on the published information, immature [DKdS05, SKK+06].

The self-consistent MPI performance requirements formulated in the third paper [3] formalizes natural expectations on the performance of various parts of MPI in relation to other parts. This is possible without an explicit performance model, which MPI does not have, and without making any absolute guarantees on performance. A weakness of course is that

MPI function	Non-local semantic checks
MPI_Finalize	
MPI_Barrier	Call consistency
MPI_Bcast	Call consistency, datasize match
MPI_Gather	Call, root consistency, datasize match
MPI_Gatherv	Call, root consistency, datasize match
MPI_Scatter	Call, root consistency, datasize match
MPI_Scatterv	Call, root consistency, datasize match
MPI_Allgather	Call, `MPI_IN_PLACE` consistency, datasize match
MPI_Allgatherv	Call, `MPI_IN_PLACE` consistency, datasize match
MPI_Alltoall	Call consistency, datasize match
MPI_Alltoallv	Call consistency, datasize match
MPI_Alltoallw	Call consistency, datasize match
MPI_Reduce	Call, root, op consistency, datasize match
MPI_Allreduce	Call, `MPI_IN_PLACE`, op consistency, datasize match
MPI_Reduce_scatter	Call, `MPI_IN_PLACE`, op consistency, datasize match
MPI_Scan	Call, op consistency, datasize match
MPI_Exscan	Call, op consistency, datasize match
MPI_Comm_dup	Call consistency
MPI_Comm_create	Call consistency, group consistency
MPI_Comm_split	Call consistency
MPI_Intercomm_merge	Call, high/low consistency
MPI_Intercomm_create	Call, local leader, tag consistency
MPI_Cart_create	Call consistency, dims consistency
MPI_Graph_create	Call consistency, graph consistency
MPI_Win_create	Call, info consistency
MPI_Win_free	Call consistency
MPI_Win_fence	Assertion consistency (`MPI_MODE_NOPRECEDE`, `MPI_MODE_NOSUCCEED`)

Table 11.1: Summary of verification performed for collective MPI operations, communication models.

MPI function	Non-local semantic checks
MPI_File_open	Call, info consistency
MPI_File_close	Call consistency
MPI_File_set_info	Call, info consistency
MPI_File_set_view	Call, info consistency, pending operations, type correctness
MPI_File_set_size	Call consistency, size match, pending operations
MPI_File_preallocate	Call consistency, size match, pending operations
MPI_File_set_atomicity	Call, flag consistency
MPI_File_seek_shared	Call consistency, offset & mode match
MPI_File_{write|read}_shared	Fileview consistency
MPI_File_{write|read}_ordered	Call consistency, fileview consistency
MPI_File_{write|read}_ordered_begin	Call consistency, pending operations, fileview consistency
MPI_File_{write|read}_ordered_end	Call consistency, operation match
MPI_File_{write|read}{_at}_all	Call consistency
MPI_File_{write|read}{_at}_all_begin	Call consistency, pending operations
MPI_File_{write|read}{_at}_all_end	Call consistency, operation match
MPI_Comm_spawn{_multiple}	Call, root consistency
MPI_Comm_accept	Call, root consistency
MPI_Comm_connect	Call, root consistency
MPI_Comm_disconnect	Call consistency

Table 11.2: Summary of verification performed for collective MPI operations, parallel I/O and process management.

it may be easier for a bad MPI implementation to fulfill the requirements than for a highly tuned one. A natural (but extreme) expectation, for instance, is that anything a user can do to improve functions of the library should already have been done! For example, there should not be an easy way of improving, say, MPI_Allreduce by putting MPI_Reduce and MPI_Bcast together. Because the various parts and communication models of MPI are semantically strongly interrelated, a lot of such rules suggest themselves naturally. For instance, more specialized collectives are not only an expressive convenience, but there is also an expectation that they might be implemented by specialized and more efficient algorithms. Most of the collectives have been formulated in such a way that this is indeed the case. By preventing a number of unnatural optimizations, it is suggested that such rules can also be an aid toward achieving some degree of *performance portability* (without a model). Coupled with a flexible, expressive and reliable benchmark, of which there is unfortunately no generally accepted instance in the MPI community, it would be possible to automate the search for gross violations of the rules in any given MPI implementation on any given system.

Chapter 12

Conclusion

It is time to summarize the contributions presented in the dissertation, the impact in the community, and to comment on future developments of MPI/SX and the related NEC MPI implementations as well as of the MPI standard itself.

12.1 Summary

The main contributions of (the author of) the work collected here are:

- Full implementation of the MPI-2 standard, in particular of the one-sided communication model with performance equivalent to that of point-to-point communication.
- The technique for more efficient handling of derived datatypes called *flattening on the fly*, and its application to for instance parallel I/O.
- A number of new and improved algorithms for collective communication and reduction operations based on point-to-point communication. New algorithms for some of the irregular collectives. Efficient implementations of these algorithms, and adaption to SMP systems with a non-homogeneous communication system.
- A non-trivial implementation of process topologies for SMP systems based on graph partitioning.
- The idea of a *verification interface* for verification of correct use of collective MPI functions in application programs.
- The idea of posing *self-consistent performance requirements* as an aid towards achieving performance portability and as performance guideline for MPI implementers.

The papers comprising the technical contribution of this dissertation are, with the exception of two, all conference contributions. Comprehensive journal versions of some of them have been submitted and/or are in preparation, but more could have deserved the time required to make them ready for journal publication. That this has not happened is due to

exactly that: limited time available. On the other hand, as is common in computer science, most parallel processing and message passing conferences have proceedings that are readily available from good natural science libraries and extensive enough to serve archival purposes for results that may after all be of more passing interest. The conferences at which (most of) the contributions were presented are considered of high to good standing, peer reviewed with good to at least reasonable scientific standards, and are relevant both general and specialist fora for presentation of MPI and message passing related work.

- EuroPVM/MPI (*European PVM/MPI Users' Group Meeting*): *the* "specialist conference" dedicated to MPI, PVM and occasionally other message passing based programming paradigms, now in its fifteenths year, taking place each September in Europe. All developers of the major MPI implementations, and influential groups and individuals attend on a regular basis. The conference is peer reviewed by experts in the field, and in recent years contributions never received less than three independent reviews. Acceptance rate is typically around or above 50%, so if a contribution is reasonable and in scope, acceptance is normally "easy". The attendance is on the order of 100 to 150.

- ACM/IEEE Supercomputing: the most prominent high-performance computing conference, held annually in the US. Supercomputing consists of an exhibition, where all major vendors, labs and research institutions in high-performance computing are present, and a broad technical conference on all aspects of high-performance computing. Competition for the technical program is high, and MPI contributions are only accepted if they are considered important by reviewers (three or more) and program committee. Supercomputing attracts several thousand attendees for the exhibit and several hundred for the technical program.

- IPDPS (*International Parallel and Distributed Processing Symposium*): a broad, IEEE hosted parallel processing conference with a long tradition (in 2002 a merger between IPPS, *International Parallel Processing Symposium*, and SPDP, *Symposium on Parallel and Distributed Processing*), with both theory and practice tracks. IPDPS is organized into a main conference of three days, and a number of one day workshops, taking place before and after the main conference. Acceptance to the main conference program is hard (in 2006 the acceptance rate was about 20%), but easier to the workshops, where programs can be of mixed quality. IPDPS usually attracts several hundred participants.

- Euro-Par: European based, broad parallel processing conference, with a peculiar organization into smaller topics with independent committees organizing reviewing and selection. An important venue for the European parallel processing community, but, partly due to its organization, quality can be wavering. Attendance is similar to IPDPS, perhaps slightly less. A recent addition to Euro-Par are thematic workshops, similar to the IPDPS format.

12.2 Impact

The most satisfying but scientifically invisible impact is that MPI/SX is production code which is used in all SX installations since 1998 (SX-4). This puts high demands on software quality, which must be bug-free and deliver the performance expected by the users. Bugs and performance problems must be resolved quickly.

The version of MPI/SX released in October 2000 was arguably the first full MPI-2 implementation. The earlier 1999 implementation for Fujitsu systems [AKL99] was lacking some dynamic process management functionality. All subsequent MPI/SX implementations (and the derivatives MPI/PC and MPI/EX for scalar systems) including the first MPI/ES implementation for the Earth Simulator of April 2002 are thus full MPI-2 implementations. For comparison `mpich2` was complete in 2004 (see for instance [TGR04] for a semi-official statement), and OpenMPI apparently first in 2006. Other vendor MPI implementations in the meantime support most of the MPI-2 standard, but often with restrictions. Which actually support the full standard is not completely clear from the available material.

Full MPI-2 support was important for the Earth Simulator, which deservedly won a number of Gordon Bell awards from November 2002 and onwards [SMSY02, YIU+02, STF+02, KTJT03]. In [YIU+02] the performance of MPI one-sided communication was mentioned as crucial.

A Master's Thesis at the University of Karlsruhe [Str04] conducts an independent, extensive evaluation of the implementation of the one-sided communication model in MPI/SX using the *SKaMPI* benchmark. On an SX-6 system it shows that high performance when compared to both peak and MPI point-to-point performance is achieved, and confirms that the hints that can be provided through assertions are exploited to improve performance.

The efficient handling of MPI derived datatypes by the *flattening on the fly* technique was pioneered by MPI/SX and was a first, necessary, important step towards making derived datatypes more generally usable by removing some of the performance penalties associated with this mechanism. The idea has been taken up by many other researchers and groups, and is in some form part of many other MPI libraries, see [WGR02, RMG03, SWP04, WWP04] to mention but a few. The incorporation of efficient datatype handling functionality into parallel I/O was also premiered by MPI/SX, and may still be a unique feature. The performance of the MPI/SX datatype handling mechanism is independently benchmarked and appreciated as highly efficient for common user datatypes in [GRR01].

The round optimal *circulant graph broadcast algorithm* was presumably the first algorithm with this property that was actually implemented for an MPI library. Recently, a different optimal algorithm, clearly inspired by this result was presented by a researcher from IBM [Jia07]. Many of the other results on collective operations are regularly cited in the relevant MPI literature.

The idea of run-time verification of correct use of collective MPI operations was well-received by the community and immediately taken up by for instance the Argonne National Laboratory group [FCLG05]. A presentation was invited to a mini-symposium at the ParCo conference in 2005.

The recently presented idea of self-consistent MPI performance requirements stimulated

much discussion at a Dagstuhl seminar on "Code Instrumentation and Modeling for Parallel Performance Analysis" and at the EuroPVM/MPI 2007 conference where it was presented at the "Outstanding Papers" session.

12.3 Future work on MPI/SX

From the implementer and algorithm developers point of view, the MPI standard has been surprisingly rich in interesting problems, and is, perhaps also surprising, far from exhausted. In addition, an MPI library belong to the basic software of a parallel system, and application developers depend on a correct, reliable, and performant implementation. So also with the users of MPI/SX and the other NEC MPI implementations. Therefore, all work presented here is ongoing and in-progress. A significant amount of time is and will be spent on elementary maintenance and tuning.

Some issues that naturally arise out of the work presented here are the development of frameworks for automatic adaptation of collective algorithms to specific system characteristics, in particular automatic dimensioning of pipeline buffer sizes, better algorithms for irregular collectives, of which there is still a general lack, investigation of data accumulation in one-sided communication, functions for copying between differently typed MPI buffers without intermediate copies (socalled *transpacking*, work-in-progress), collective algorithms for multi-ported systems which will become more relevant in the coming years (for instance for the NEC SX-9 announced at ACM/IEEE Supercomputing 2007), and support for heterogeneous systems, which will also become an important issue in the coming years.

Compared to many other MPI libraries MPI/SX is a very mature implementation. In [TG07] several areas in which MPI libraries in general are in need of improvement are discussed. These include scalability (to tens or hundred thousands of processes), point-to-point latency, collective performance, datatype performance, one-sided communication performance, multi-thread support and performance, fault tolerance and debugging and performance tool support. As discussed and demonstrated by the research collected in this dissertation, these areas all have been and continue to be actively addressed by the MPI/SX implementation.

12.4 Lessons learned

MPI/SX is a large piece of system software of several 100,000 lines of code, and the demands on correctness, performance, and completeness are extremely high. However, due to MPI's economy of concepts, the same machinery (e.g. basic point-to-point functionality, datatype handling functions, communicators and group management functions, low-level synchronization mechanisms for shared-memory programming) is used again and again for various parts of the standard, and thus needs to be tested only once. Or the other way round, since the same machinery is used in many different contexts, it is tested under many different conditions, sometimes in some for which it was not originally developed. This strong consistency

aids strongly in securing correctness, and is a further witness to the strong design of MPI. Also because of the economy of concepts, and because MPI/SX was originally built on the MPICH design [GLDS96], it has been possible to develop MPI/SX software in a small team (3-5 person strong).

Testing of the software is essential and indispensable, and done mostly by means of external, small MPI test programs by the developer (and sometimes with reluctance). To ensure completeness, that is that all parts of the implementation have been exercised, is difficult by this method, and requires much care. External testing that the requirements of the standard are fulfilled for the individual MPI functions in isolation is easier, but no small task (also in pure running time, since the library has to be tested for varying system sizes and configurations) and could have been (and for MPI/SX and MPI/EX is to some extent) done by another team than the development team. It would be of extreme value for the MPI community to have an agreed upon, extensive, external test suite for the whole standard. This could also help clarifying some of the points in the standard which are open to debate. There is unfortunately no such test suite in the community.

Performance is mostly ascertained by fixed benchmark programs, and in a product with a long history like MPI/SX, monotonically non-decreasing improvements for each new version are important and expected by users. This is coupled with feedback from applications, which could (and should) be much more systematic. As was the case for a community test suite, there is unfortunately no agreed upon, common, exhaustive, sound, flexible benchmark for measuring the performance of MPI functions. This is a deep lack.

12.5 Outlook on the MPI standard

Despite MPI often being criticized as being too explicit, low-level, large, complicated, tedious and so on, MPI has been extremely successful. By establishing itself as a *de facto* standard from early on, MPI has been a major, driving force behind the advancement of high- and (with the advent of affordable SMP clusters in the late 1990ties) medium-performance parallel processing. An important feature has been the fact that MPI was designed to build libraries, and through this, and the relatively few key concepts of MPI that has to be learned to use the interface, many of the arguments above against MPI loose force.

Despite many attempts and proposals for replacing MPI, there is consensus that MPI will survive – healthily – as a main programming model in the years to come, even for the PetaFlop systems with 100,000 of processors that are currently being planned in the US and in Japan and that will become operational perhaps already by the end of this decade. The designed alternatives, for instance the languages in the HPCS initiative [LY07] will not be ready or accepted enough to seriously challenge MPI (for better or for worse).

This naturally raises the question of new developments for and additions to MPI, in short the question of whether there is a need for/will be an MPI-3.

Over the years and since the MPI-2 additions have become implemented and are being used in more and more applications, a number of inaccuracies and other problems in the standard have been detected. There is a clear need for these to be addressed and fixed, and

the MPI Forum is currently active in doing this. Minor corrections have already been adopted in the consolidated MPI 2.1 standard [MPI08], and issues of more consequence are being considered for MPI 2.2. Some issues were raised by the author in the preceding chapters, some of them being corrections and clarifications in the process topology functionality, possibly with new, orthogonal functionality to give information from the MPI library back to a mesh partitioner, a regular MPI_Reduce_scatter_block collective, built-in reduction operators for segmented scans, and other such things. An important issue to address is the typing of `count` arguments as integers (`int`) which is already problematic in applications dealing with amounts of data larger than 2GBytes. Here, it is a problem with an accepted standard that function interfaces cannot simply be changed, and the sometimes adopted MPI method of *deprecating* functions will hardly work (too many functions to deprecate).

A more far-ranging issue is whether it is desirable (or necessary) to introduce collectives that are non-blocking, in order to permit more overlap between computation and communication in applications relying heavily on collectives. The MPI *Journal of Development* already has a limited proposal for split collectives (as they now exist for parallel I/O), but this is rightly criticized in [HGLR07, HKG+07] as limited, and a proposal for more genuinely asynchronous, non-blocking collectives is given.

For very large, possibly heterogeneous systems it is questionable whether the illusion of a flat system is sustainable, so better, more flexible resource and mapping control may be needed. The existing process topology functionality could be the starting point for more radical extensions in this direction, but it could also be that such issues are better handled outside of MPI.

A fraction of users, system administrators and developers who find MPI too powerful, too large, and therefore too inefficient (in terms of being able to exploit hardware resources) often ask for a "light-weight" MPI. There are different schools in this way of thinking, few concrete proposals, and it is not likely that MPI will change much in this direction. A counter argument is that some of the allegedly expensive features of MPI like derived datatypes and communicators do not cause much overhead (in a good implementation) if they are not used.

Finally, the largest, most frequently and loudly discussed issue is that of fault-tolerance. With the jump from thousands of processors to the 256K processors of the IBM Blue Gene/L machine it was widely believed that the mean time between hardware (or software) failures would be so small that it would be impossible to run large applications (requiring several hours of computing time even on this scale of system) without some support for software fault-tolerance. The experience with the Earth Simulator, and later with the Blue Gene showed, surprisingly, that these concerns were exaggerated. However, the issue remains and there is a substantial drive for extending MPI in some aspects to provide for fault-tolerance. It is not clear how well-founded the concerns or the current proposals are. Regardless of which model is chosen, there will be consequences for the user, either in the form of (substantial) performance penalties or in the form of having to program applications with fault-tolerance in mind. Also, ways of providing fault-recovery by e.g. checkpointing are well incorporated in current practice and do not require (much) MPI support. In [GL04] it is showed what is already possible in terms of fault-tolerance within MPI as it is. Some further pointers to

the ongoing discussion and proposals are [BBC+02, BCH+03, BLKC03, HLH+06, BDSB04, FD00, FBD01, FAB+05].

Bibliography

[AG06] Denis Trystram Alfredo Goldman, Joseph G. Peters. Exchanging messages of different sizes. *Journal of Parallel and Distributed Computing*, 66(1):1–18, 2006.

[AISS97] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris J. Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[AKL99] Noboru Asai, Thomas Kentemich, and Pierre Lagier. MPI-2 implementation on Fujitsu generic message passing kernel. In *Supercomputing*, 1999. `http://www.supercomp.org/sc99/proceedings/techpap.htm#mpi`.

[ASW05] Werner Augustin, Marc-Oliver Straub, and Thomas Worsch. Benchmarking one-sided communication with SKaMPI 5. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 12th European PVM/MPI Users' Group Meeting*, volume 3666 of *Lecture Notes in Computer Science*, pages 301–308. Springer-Verlag, 2005.

[BAH+05] Jon Beecroft, David Addison, David Hewson, Moray McLaren, Duncan Roweth, Fabrizio Petrini, and Jarek Nieplocha. QsNetII: Defining high-performance network design. *IEEE Micro*, 25(4):34–47, 2005.

[BBB+94] Vasanth Bala, Jehoshua Bruck, Raymond Bryant, Robert Cypher, Peter de Jong, Pablo Elustondo, Daniel D. Frye, Alex Ho, Ching-Tien Ho, Gail Irwin, Shlomo Kipnis, Richard D. Lawrence, and Marc Snir. The IBM external user interface for scalable parallel systems. *Parallel Computing*, 20(4):445–462, 1994.

[BBC+95] Vasanth Bala, Jehoshua Bruck, Robert Cypher, Pablo Elustondo, Alex Ho, Ching-Tien Ho, Shlomo Kipnis, and Marc Snir. CCL: A portable and tunable collective communications library for scalable parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):154–164, 1995.

[BBC+02] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cécile Germain, Thomas Hérault, Pierre Lemarinier, Oleg Lodygensky,

Frédéric Magniette, Vincent Néri, and Anton Selikhov. MPICH-V: toward a scalable fault tolerant MPI for volatile nodes. In *ACM/IEEE Supercomputing*, pages 1–18, 2002.

[BBC+03] Christian Bell, Dan Bonachea, Yannick Cote, Jason Duell, Paul Hargrove, Parry Husbands, Costin Iancu, Michael L. Welcome, and Katherine A. Yelick. An evaluation of current high-performance networks. In *17th International Parallel and Distributed Processing Symposium (IPDPS03)*, page 28, 2003.

[BBG+87] J. Boyle, R. Butler, B. Glickfeld, T. Disz, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programming for Parallel Processors*. Holt, Winston, and Reinhart, 1987.

[BCD+96] Jehoshua Bruck, Luc De Coster, Natalie Dewulf, Ching-Tien Ho, and Rudy Lauwereins. On the design and implementation of broadcast and global combine operations using the postal model. *IEEE Transactions on Parallel and Distributed Systems*, 7(3):256–265, 1996.

[BCH+03] Aurelien Bouteiller, Franck Cappello, Thomas Hérault, Géraud Krawezik, Pierre Lemarinier, and Frédéric Magniette. MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *ACM/IEEE Supercomputing*, page 25, 2003.

[BD04] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *International Journal of High Performance Computing and Networking*, 1(1/2/3):91–99, 2004.

[BDSB04] Rajanikanth Batchu, Yoginder S. Dandass, Anthony Skjellum, and Murali Beddhu. MPI/FT: A model-based approach to low-overhead fault tolerant message-passing middleware. *Cluster Computing*, 7(4):303–315, 2004.

[BGP+94] Mike Barnett, Satya Gupta, David G. Payne, Lance Schuler, Robert van de Geijn, and Jerell Watts. Building a high-performance collective communication library. In *Supercomputing'94*, pages 107–116, 1994.

[BGST03] Surendra Byna, William D. Gropp, Xian-He Sun, and Rajeev Thakur. Improving the performance of MPI derived datatypes by optimizing memory-access cost. In *IEEE International Conference on Cluster Computing (CLUSTER 2003)*, pages 412–419, 2003.

[BHK+97] Jehoshua Bruck, Ching-Tien Ho, Schlomo Kipnis, Eli Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, 1997.

[BHP+99] Gianfranco Bilardi, Kieran T. Herley, Andrea Pietracaprina, Geppino Pucci, and Paul G. Spirakis. BSP versus LogP. *Algorithmica*, 24(3–4):405–422, 1999.

[BHS+02] Georg Bißeling, Hans-Christian Hoppe, Alexander V. Supalov, Pierre Lagier, and Jean Latour. Fujitsu MPI-2: Fast locally, reaching globally. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 9th European PVM/MPI Users' Group Meeting*, volume 2474 of *Lecture Notes in Computer Science*, pages 401–409. Springer-Verlag, 2002.

[BIC05] Massimo Bernaschi, Giulio Iannello, and Saverio Crea. Experimental results about MPI collective communication operations. *Parallel Processing Letters*, 15(1–2):223–236, 2005.

[BIL02] Massimo Bernaschi, Giulio Iannello, and Mario Lauria. Efficient implementation of reduce-scatter in MPI. In *10th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP 2002)*, pages 301–308, 2002.

[Bis04] Rob H. Bisseling. *Parallel Scientific Computation. A Structured Approach using BSP and MPI.* Oxford University Press, 2004.

[BJvOR03] Olaf Bonorden, Ben H. H. Juurlink, Ingo von Otte, and Ingo Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187–207, 2003.

[BL94] Ralph Butler and Ewing L. Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20(4):547–564, 1994.

[Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, 1990.

[BLKC03] Aurelien Bouteiller, Pierre Lemarinier, Graud Krawezik, and Franck Cappello. Coordinated checkpoint versus message log for fault tolerant MPI. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 242–250, 2003.

[BM00] S. Booth and E. Mourão. Single sided MPI implementations for SUN MPI. In *Supercomputing*, 2000. `http://www.sc2000.org/proceedings/techpapr/index.htm#01`.

[BNBH+95] Amotz Bar-Noy, Jehoshua Bruck, Ching-Tien Ho, Schlomo Kipnis, and Baruch Schieber. Computing global combine operations in the multiport postal model. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):896–900, 1995.

[BNK94a] Amotz Bar-Noy and Shlomo Kipnis. Broadcasting multiple messages in simultaneous send/receive systems. *Discrete Applied Mathematics*, 55(2):95–105, 1994.

[BNK94b] Amotz Bar-Noy and Shlomo Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. *Mathematical Systems Theory*, 27(5):431–452, 1994.

[BNK97] Amotz Bar-Noy and Shlomo Kipnis. Multiple message broadcasting in the postal model. *Networks*, 29(1):1–10, 1997.

[BNKS00] Amotz Bar-Noy, Shlomo Kipnis, and Baruch Schieber. Optimal multiple message broadcasting in telephone-like communication systems. *Discrete Applied Mathematics*, 100(1–2):1–15, 2000.

[BPS01] Tiago Baptista, Hernâni Pedroso, and João Gabriel Silva. The implementation of one-sided communications for WMPI II. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 8th European PVM/MPI Users' Group Meeting*, volume 2131 of *Lecture Notes in Computer Science*, pages 61–68. Springer-Verlag, 2001.

[BSTG06] Surendra Byna, Xian-He Sun, Rajeev Thakur, and William Gropp. Automatic memory optimizations for improving MPI derived datatype performance. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[CCZ07] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21:291–312, 2007.

[CDK+01] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2001.

[CDMC+05] Cristian Coarfa, Yuri Dotsenko, John M. Mellor-Crummey, Franois Cantonnet, Tarek A. El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel G. Chavarría-Miranda. An evaluation of global address space languages: Co-array Fortran and Unified Parallel C. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 36–47, 2005.

[CE00] Franck Cappello and Daniel Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks. In *Supercomputing*, 2000. See `http://www.sc2000.org/proceedings/techpapr/index.htm#04`.

[CFMR05] Franck Cappello, Pierre Fraigniaud, Bernard Mans, and Arnold L. Rosenberg:. An algorithmic model for heterogeneous hyper-clusters: rationale and experience. *International Journal of Foundations of Computer Science*, 16(2):195–215, 2005.

[CG89] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[CGMS94] Nicholas Carriero, David Gelernter, Timothy G. Mattson, and Andrew H. Sherman. The Linda alternative to message-passing systems. *Parallel Computing*, 20(4):633–655, 1994.

[CHHW94] Robin Calkin, Rolf Hempel, H.-C. Hoppe, and P. Wypior. Portable programming with the PARMACS message-passing library. *Parallel Computing*, 20(4):615–632, 1994.

[CHPvdG07] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert A. van de Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[CK96] Robert Cypher and Smaragda Konstantinidou. Bounds on the efficiency of message-passing protocols for parallel computers. *SIAM Journal on Computing*, 25(5):1082–1104, 1996.

[CKP+96] David E. Culler, Richard M. Karp, David Patterson, Abhijit Sahay, Eunice E. Santos, Klaus Erik Schauser, Ramesh Subramonian, and Thorsten von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, 1996.

[CSG99] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture. A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1999.

[CvdGGT06] Ernie Chan, Robert A. van de Geijn, William Gropp, and Rajeev Thakur. Collective communication on architectures that support simultaneous communication over multiple links. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 2–11, 2006.

[Dar01] Frederica Darema. The SPMD model : Past, present and future. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 8th European PVM/MPI Users' Group Meeting*, volume 2131 of *Lecture Notes in Computer Science*, page 1. Springer-Verlag, 2001.

[Dem97] Erik D. Demaine. A threads-only MPI implementation for the development of parallel programs. In *Proceedings of the 11th International Symposium on High Performance Computing Systems (HPCS'97)*, pages 153–163, 1997.

[DFC+02] F. Dehne, A. Ferreira, E. Cáceres, S. W. Song, and A. Roncato. Efficient parallel graph algorithms for coarse-grained multicomputers and BSP. *Algorithmica*, 33:183–200, 2002.

[DGNP88] Frederica Darema, David A. George, V. Alan Norton, and Gregory F. Pfister. A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Computing*, 7(1):11–24, 1988.

[DKdS05] Jayant DeSouza, Bob Kuhn, and Bronis R. de Supinski. Automated, scalable debugging of MPI programs with Intel Message Checker. In *Second International Workshop on Software Engineering for High Performance Computing System Applications in conjunction with 27th International Conference on Software Engineering*, 2005.

[DM98] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computer Science and Engineering*, 5(1):46–55, 1998.

[DOSW96] Jack Dongarra, Steve W. Otto, Marc Snir, and David W. Walker. A message passing standard for MPP and workstations. *Communications of the ACM*, 39(7):84–90, 1996.

[DSU04] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[DT04] William James Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, 2004.

[DYN03] José Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks*. Morgan Kaufmann Publishers, revised printing edition, 2003.

[EGCSY05] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared Memory Programming*. Wiley, 2005.

[FAB+05] Graham E. Fagg, Thara Angskun, George Bosilca, Jelena Pješivac-Grbović, and Jack Dongarra. Scalable fault tolerant MPI: Extending the recovery algorithm. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 12th European PVM/MPI Users' Group Meeting*, volume 3666 of *Lecture Notes in Computer Science*, pages 67–75. Springer-Verlag, 2005.

[FBD01] Graham E. Fagg, Antonin Bukovsky, and Jack Dongarra. HARNESS and fault tolerant MPI. *Parallel Computing*, 27(11):1479–1495, 2001.

[FCLA06] M. Farreras, T. Cortes, J. Labarta, and G. Almasi. Scaling MPI to short-memory MPPs such as BG/L. In *ACM International Conference on Supercomputing (ICS)*, pages 209–217, 2006.

[FCLG05] Chris Falzone, Anthony Chan, Ewing Lusk, and William Gropp. Collective error detection for MPI collective operations. In *Recent Advances in Parallel*

Virtual Machine and Message Passing Interface. 12th European PVM/MPI Users' Group Meeting, volume 3666 of *Lecture Notes in Computer Science*, pages 138–147. Springer-Verlag, 2005.

[FD00] Graham E. Fagg and Jack Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 7th European PVM/MPI Users' Group Meeting*, volume 1908 of *Lecture Notes in Computer Science*, pages 346–353. Springer-Verlag, 2000.

[FK94] Jon Flower and Adam Kolawa. Express is not just a message passing system. Current and future directions in express. *Parallel Computing*, 20(4):597–614, 1994.

[FL94] Pierre Fraigniaud and Emmanuel Lazard. Methods and problems of communication in usual networks. *Discrete Applied Mathematics*, 53(1–3):79–133, 1994.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21:948–960, 1072.

[FM82] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *19th ACM/IEEE Design Automation Conference (DAC)*, pages 175–181, 1982.

[FVD00] Graham E. Fagg, Sathish S. Vadhiyar, and Jack Dongarra. ACCT: Automatic collective communications tuning. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 7th European PVM/MPI Users' Group Meeting*, volume 1908 of *Lecture Notes in Computer Science*, pages 354–362. Springer-Verlag, 2000.

[FW78] Steven Fortune and James Wyllie. Parallelism in random access machines. In *10th Annual ACM Symposium on Theory of Computing (STOC)*, pages 114–118, 1978.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[GEL+06] Assefaw Hadish Gebremedhin, Mohamed Essaïdi, Isabelle Guérin Lassous, Jens Gustedt, and Jan Arne Telle. PRO: A model for the design and analysis of efficient and scalable parallel algorithms. *Nordic Journal of Computing*, 13:215–239, 2006.

[GFB+04] 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 *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 97–104. Springer-Verlag, 2004.

[GFD03] Edgar Gabriel, Graham E. Fagg, and Jack Dongarra. Evaluating the performance of MPI-2 dynamic communicators and one-sided communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 10th European PVM/MPI Users' Group Meeting*, volume 2840 of *Lecture Notes in Computer Science*, pages 88–97. Springer-Verlag, 2003.

[GHLL+98] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI – The Complete Reference*, volume 2, The MPI Extensions. MIT Press, 1998.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979. With an addendum, 1991.

[GKKG03] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing*. Addison-Wesley, second edition, 2003.

[GL04] William Gropp and Ewing Lusk. Fault tolerance in message passing interface programs. *International Journal of High Performance Computing Applications*, 18(3):363–372, 2004.

[GLDS96] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable imlementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[GLR+99] Mark Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, and Thanasis Tsantilas. Portable and efficient parallel computing using the BSP model. *IEEE Transactions on Computers*, 48(7):670–689, 1999.

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994. Second printing, 1995.

[GLT99] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, 1999.

[GMPD98] Delphine Stéphanie Goujon, Martial Michel, Jasper Peeters, and Judith Ellen Devaney. AutoMap and AutoLink: Tools for communicating complex and dynamic data-structures using MPI. In *Network-Based Parallel Computing. Communication, Architecture, and Applications*, volume 1362 of *Lecture Notes in Computer Science*, pages 98–109. Springer-Verlag, 1998.

[Gor04] Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Transactions on Programming Languages and Systems*, 26(1):47–56, 2004.

[GRM04] William D. Gropp, Robert Ross, and Neill Miller. Providing efficient I/O redundancy in MPI environments. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 77–86. Springer-Verlag, 2004.

[Gro00] William Gropp. Runtime checking of datatype signatures in MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 7th European PVM/MPI Users' Group Meeting*, volume 1908 of *Lecture Notes in Computer Science*, pages 160–167. Springer-Verlag, 2000.

[GRR01] Edgar Gabriel, Michael Resch, and Roland Rühle. Implementing and benchmarking derived datatypes in metacomputing. In *High-Performance Computing and Networks (HPCN'2001)*, volume 2110 of *Lecture Notes in Computer Science*, pages 493–502. Springer-Verlag, 2001.

[GT05] William D. Gropp and Rajeev Thakur. An evaluation of implementation options for MPI one-sided communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 12th European PVM/MPI Users' Group Meeting*, volume 3666 of *Lecture Notes in Computer Science*, pages 415–424. Springer-Verlag, 2005.

[GT07a] William Gropp and Rajeev Thakur. Thread-safety in an MPI implementation: Requirements and analysis. *Parallel Computing*, 33(9):595–604, 2007.

[GT07b] William D. Gropp and Rajeev Thakur. Revealing the performance of MPI RMA implementations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 14th European PVM/MPI Users' Group Meeting*, volume 4757 of *Lecture Notes in Computer Science*, pages 272–280. Springer-Verlag, 2007.

[Han98] Per Brinch Hansen. An evaluation of the message-passing interface. *SIGPLAN Notices*, 33(3):65–72, 1998.

[Hat98] Takao Hatazaki. Rank reordering strategy for MPI topology creation functions. In *5th European PVM/MPI User's Group Meeting*, volume 1497 of *Lecture Notes in Computer Science*, pages 188–195. Springer-Verlag, 1998.

[Hen00] D. S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Supercomputing*, 2000. See `http://www.sc2000.org/proceedings/techpapr/index.htm#04`.

[HGLR07] Torsten Hoefler, Peter Gottschling, Andrew Lumsdaine, and Wolfgang Rehm. Optimizing a conjugate gradient solver with non-blocking collective operations. *Parallel Computing*, 33(9):624–633, 2007.

[HHL88] Sandra M. Hedetniemi, T. Hedetniemi, and Arthur L. Liestman. A survey of gossiping and broadcasting in communication networks. *Networks*, 18:319–349, 1988.

[HHMW94] Rolf Hempel, Anthony J. G. Hey, Oliver McBryan, and David W. Walker. Special issue: Message passing interfaces. *Parallel Computing*, 20(4):415–678, 1994.

[HI00] Roger Hillson and Michal Iglewski. C++2MPI: A software tool for automatically generating MPI datatypes from C++ classes. In *Parallel Computing in Electrical Engineering (PARELEC'00)*, pages 13–17, 2000.

[Hig93] High Performance Fortran Forum. High Performance Fortran language specification (version 1.0). *Scientific Programming*, 2(1/2):1–170, 1993.

[HKG+07] Torsten Hoefler, Prabhanjan Kambadur, Richard L. Graham, Galen M. Shipman, and Andrew Lumsdaine. A case for standard non-blocking collective operations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting*, volume 4757 of *Lecture Notes in Computer Science*, pages 125–134. Springer-Verlag, 2007.

[HLH+06] William Hoarau, Pierre Lemarinier, Thomas Hérault, Eric Rodriguez, Sébastien Tixeuil, and Franck Cappello. FAIL-MPI: How fault-tolerant is fault-tolerant MPI? In *IEEE International Conference on Cluster Computing (CLUSTER)*, 2006.

[HMLR07] Torsten Hoefler, Torsten Mehlan, Andrew Lumsdaine, and Wolfgang Rehm. Netgauge: A network performance measurement framework. In *High Performance Computing and Communications (HPPC)*, pages 659–671, 2007.

[HMS+98] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, 1998.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hoa91] C. A. R. Hoare. The transputer and occam: a personal story. *Concurrency: practice and experience*, 3(4):249–264, August 1991.

[Hoc94] Roger W. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20(3):389–398, 1994.

[HRZ98] Rolf Hempel, Hubert Ritzdorf, and Falk Zimmermann. Efficient message passing interface implementations for NEC parallel computers. *NEC Research & Development*, 39(4):408–413, 1998.

[HSN81] Kai Hwang, Shun-Piao Su, and Lionel M. Ni. Vector computer architectures and processing techniques. *Advances in Computers*, 20:115–197, 1981.

[HV05] Hans Henrik Happe and Brian Vinter. Improving TCP/IP multicasting with message segmentation. In *Communicating Process Architectures (CPA 2005)*, volume 63 of *Concurrent Systems Engineering Series*, pages 155–163, 2005.

[HW99] Rolf Hempel and David W. Walker. The emergence of the MPI message passing standard for parallel computing. *Computer Standards & Interfaces*, 21:51–62, 1999.

[Ian97] Giulio Iannello. Efficient algorithms for the reduce-scatter operation in LogGP. *IEEE Transactions on Parallel and Distributed Systems*, 8(9):970–982, 1997.

[INM88] INMOS Limited. 1988.

[JáJ92] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[JH89] S. Lennart Johnsson and Ching-Tien Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, 38(9):1249–1268, 1989.

[Jia07] Bin Jia. Process cooperation in multiple message broadcast. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 14th European PVM/MPI Users' Group Meeting*, volume 4757 of *Lecture Notes in Computer Science*, pages 27–35. Springer-Verlag, 2007.

[JLJ+04] Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, Dhabaleswar K. Panda, Darius Buntinas, Rajeev Thakur, and William D. Gropp. Efficient implementation of MPI-2 passive one-sided communication on InfiniBand clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 68–76. Springer-Verlag, 2004.

[JW96] Ben H. H. Juurlink and Harry A. G. Wijshoff. Communication primitives for BSP computers. *Information Processing Letters*, 58(6):303–310, 1996.

[KBMR04] Bettina Krammer, Katrin Bidmon, Matthias S. Müller, and Michael M. Resch. MARMOT: An MPI analysis and checking tool. In *Parallel Computing: Software Technology, Algorithms, Architectures and Applications, (PARCO)*, volume 13 of *Advances in Parallel Computing*, pages 493–500, 2004.

[KBV00] Thilo Kielmann, Henri E. Bal, and Kees Verstoep. Fast measurement of LogP parameters for message passing platforms. In *IPDPS 2000 Workshops*, volume 1800 of *Lecture Notes in Computer Science*, pages 1176–1183. Springer-Verlag, 2000.

[KC95] Oh-Heum Kwon and Kyung-Yong Chwa. Multiple message broadcasting in communication networks. *Networks*, 26:253–261, 1995.

[KK03] George Em Karniadakis and Robert M. Kirby. *Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and their Implementation*. Cambridge University Press, 2003.

[KKT01] Jörg Keller, Christoph W. Keßler, and Jesper Larsson Träff. *Practical PRAM Programming*. John Wiley & Sons, 2001.

[KL70] Brian W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49:291–307, 1970.

[KMR04a] Bettina Krammer, Matthias S. Müller, and Michael M. Resch. MPI application development using the analysis tool MARMOT. In *International Conference on Computational Science (ICCS)*, volume 3038 of *Lecture Notes in Computer Science*, pages 464–471. Springer-Verlag, 2004.

[KMR04b] Bettina Krammer, Matthias S. Müller, and Michael M. Resch. MPI I/O analysis and error detection with MARMOT. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 242–250. Springer-Verlag, 2004.

[KMR06] Bettina Krammer, Matthias S. Müller, and Michael M. Resch. Runtime checking of MPI applications with MARMOT. In *Parallel Computing: Current & Future Issues of High-End Computing (ParCo 2005)*, volume 33 of *NIC Series*, pages 893–900. John von Neuman Institute for Computing, Central Institute for Applied Mathematics, Forschungszentrum Jülich, 2006.

[KR06] Bettina Krammer and Michael M. Resch. Correctness checking of MPI one-sided communication using marmot. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI Users' Group Meeting*, volume 4192 of *Lecture Notes in Computer Science*, pages 105–114. Springer-Verlag, 2006.

[KRS90] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. Efficient parallel algorithms for graph problems. *Algorithmica*, 5:43–64, 1990.

[KSOS05] Shoaib Kamil, John Shalf, Leonid Oliker, and David Skinner. Understanding ultra-scale application communication requirements. In *IEEE International Workload Characterization Symposium (ISSWC)*, pages 178–187, 2005.

[KTJT03] Dimitri Komatitsch, Seiji Tsuboi, Chen Ji, and Jeroen Tromp. A 14.6 billion degrees of freedom, 5 TeraFLOPS, 2.5 Terabyte earthquake simulation on the Earth Simulator. In *Supercomputing*, 2003.

[LBFD05] Julien Langou, George Bosilca, Graham E. Fagg, and Jack Dongarra. Hash functions for datatype signatures in MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 12th European PVM/MPI Users' Group Meeting*, volume 3666 of *Lecture Notes in Computer Science*, pages 76–83. Springer-Verlag, 2005.

[LCC+03] Glenn R. Luecke, Hua Chen, James Coyle, Jim Hoekstra, Marina Kraeva, and Yan Zou. MPI-CHECK: a tool for checking fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15(2):93–100, 2003.

[LWP02] Wenheng Liu, Cho-Li Wang, and Viktor K. Prasanna. Portable and scalable algorithm for irregular all-to-all communication. *Journal of Parallel and Distributed Computing*, 62:1493–1526, 2002.

[LY07] Ewing L. Lusk and Katherine A. Yelick. Languages for high-productivity computing: the DARPA HPCS language project. *Parallel Processing Letters*, 17(1):89–102, 2007.

[Mar02] John Markoff. Japanese computer is world's fastest, as U.S. falls back. *New York Times*, CLI(52094), April 2002.

[McB94] Oliver A. McBryan. An overview of message passing environments. *Parallel Computing*, 20(4):417–444, 1994.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Mil85] Robin Milner. Lectures on a calculus for communicating systems. In *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 197–220. Springer-Verlag, 1985.

[MP93] Ernst W. Mayr and C. Greg Plaxton. Pipelined parallel prefix computations, and sorting on a pipelined hypercube. *Journal of Parallel and Distributed Computing*, 17:374–380, 1993.

[MPI08] MPI Forum. *MPI: A Message-Passing Interface Standard. Version 2.1*, September 4th 2008. `www.mpi-forum.org`.

[MPSvdG95] Prasenjit Mitra, David G. Payne, Lance Schuler, and Robert van de Geijn. Fast collective communication libraries, please. In *Intel Supercomputer Users' Group Meeting*, 1995.

[MS98] José Marinho and João Gabriel Silva. WMPI – message passing interface for Win32 clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 5th European PVM/MPI Users' Group Meeting*, volume 1497 of *Lecture Notes in Computer Science*, pages 113–120. Springer-Verlag, 1998.

[MS99] Fernando Elson Mourão and João Gabriel Silva. Implementing MPI's one-sided communications in WMPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 6th European PVM/MPI Users' Group Meeting*, volume 1697 of *Lecture Notes in Computer Science*, pages 231–238. Springer-Verlag, 1999.

[MSA+07] José E. Moreira, Valentina Salapura, George Almási, Charles Archer, Ralph Bellofatto, Peter Bergner, Randy Bickford, Matthias A. Blumrich, José R. Brunheroto, Arthur A. Bright, Michael Brutman, José G. Casta nos, Dong Chen, Paul Coteus, Paul Crumley, Sam Ellis, Thomas Engelsiepen, Alan Gara, Mark Giampapa, Tom Gooding, Shawn Hall, Ruud A. Haring, Roger Haskin, Philip Heidelberger, Dirk Hoenicke, Todd Inglett, Gerard V. Kopcsay, Derek Lieber, David Limpert, Patrick McCarthy, Mark Megerian, Michael Mundy, Martin Ohmacht, Jeff Parker, Rick A. Rand, Don Reed, Ramendra K. Sahoo, Alda Sanomiya, Richard Shok, Brian Smith, Gordon G. Stewart, Todd Takken, Pavlos Vranas, Brian Wallenfelt, Michael Blocksome, and Joe Ratterman. The Blue Gene/L supercomputer: A hardware and software story. *International Journal of Parallel Programming*, 35(3):181–206, 2007.

[MSD99] Martial Michel, Andre Schaff, and Judith Devaney. Managing data-types: the CORBA approach and AutoLink, an MPI solution. In *Third MPI Developer's and User's Conference (MPIDC'99)*, pages 143–150, 1999.

[NH97] Jarek Nieplocha and Robert J. Harrison. Shared memory programming in metacomputing environments: The global array approach. *The Journal of Supercomputing*, 11(2):119–136, 1997.

[NHL94] Jarek Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: a portable "shared-memory" programming model for distributed memory computers. In *ACM/IEEE Supercomputing*, pages 340–349, 1994.

[OCC+04] Leonid Oliker, Andrew Canning, Jonathan Carter, John Shalf, and Stephane Ethier. Scientific computations on modern parallel vector systems. In *Supercomputing*, 2004. `http://www.sc-conference.org/sc2004/`.

[OCC+05] Leonid Oliker, Andrew Canning, Jonathan Carter, John Shalf, David Skinner, Stphane Ethier, Rupak Biswas, M. Jahed Djomehri, and Rob F. Van der Wijngaart. Performance evaluation of the SX-6 vector architecture for scientific computations. *Concurrency – Practice and Experience*, 17(1):69–93, 2005.

[OCS+03] Leonid Oliker, Jonathan Carter, John Shalf, David Skinner, Stephane Ethier, Rupak Biswas, Jahed Djomehri, and Rob van der Wijngaart. Evaluation of cache-based superscalar and cacheless vector architectures for scientific computations. In *Supercomputing*, 2003. `http://www.sc-conference.org/sc2003/`.

[Pac97] Peter Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann, 1997.

[PGAB+07] Jelena Pješivac-Grbović, Thara Angskun, George Bosilca, Graham E. Fagg, Edgar Gabriel, and Jack Dongarra. Performance analysis of MPI collective operations. *Cluster Computing*, 10(2):127–143, 2007.

[PGFA+07] Jelena Pješivac-Grbović, Graham E. Fagg, Thara Angskun, George Bosilca, and Jack Dongarra. MPI collective algorithm selection and quadtree encoding. *Parallel Computing*, 33(9):613–623, 2007.

[PH96] David A. Patterson and John L. Hennessy. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, second edition, 1996.

[PH04] David A. Patterson and John L. Hennessy. *Computer Organization and Design.* Morgan Kaufmann Publishers, third edition, 2004.

[PS00] Hernâni Pedroso and João Gabriel Silva. The WMPI library evolution: Experience with MPI development for windows environments. In *Euro-Par 2000 Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 1157–1164. Springer-Verlag, 2000.

[QA06] Ying Qian and Ahmad Afsahi. Efficient RDMA-based multi-port collectives on multi-rail QsNet^{II} clusters. In *Workshop on Communication Architecture for Clusters (CAC), 20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, page 273, 2006.

[QA07a] Ying Qian and Ahmad Afsahi. High performance RDMA-based multi-port all-gather on multi-rail QsNetII. In *International Symposium on High Performance Computing Systems and Applications (HPCS 2007)*, page 3, 2007.

[QA07b] Ying Qian and Ahmad Afsahi. RDMA-based and SMP-aware multi-port all-gather on multi-rail QsNet^{II} SMP clusters. In *International Conference on Parallel Processing (ICPP 2007)*, page 48, 2007.

[Qui03] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP.* McGraw-Hill, 2003.

[Rab99a] Rolf Rabenseifner. Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512. In *Third MPI Developer's and User's Conference (MPIDC'99)*, pages 77–85, 1999.

[Rab99b] Rolf Rabenseifner. Automatic profiling of MPI applications with hardware performance counters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 6th European PVM/MPI Users' Group Meeting*, volume 1697 of *Lecture Notes in Computer Science*, pages 35–42. Springer-Verlag, 1999.

[Rab02] Rolf Rabenseifner. Communication and optimization aspects on hybrid architectures. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 9th European PVM/MPI Users' Group Meeting*, volume 2474 of *Lecture Notes in Computer Science*, pages 410–420. Springer-Verlag, 2002.

[Rei93] John H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers, 1993.

[RMG03] Robert Ross, Neill Miller, and William D. Gropp. Implementing fast and reusable datatype processing. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 10th European PVM/MPI Users' Group Meeting*, volume 2840 of *Lecture Notes in Computer Science*, pages 404–413. Springer-Verlag, 2003.

[RN07] John Reid and Robert W. Numrich. Co-arrays in the next Fortran standard. *Scientific Programming*, 15(1):9–26, 2007.

[RP06] Eric Renault and Christian Parrot. MPI pre-processor: Generating MPI derived datatypes from C datatypes automatically. In *International Conference on Parallel Processing Workshops (ICPP)*, pages 248–256, 2006.

[RSPM98] Ralf Reussner, Peter Sanders, Lutz Prechelt, and Matthias Müller. SKaMPI: A detailed, accurate MPI benchmark. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 5th European PVM/MPI Users' Group Meeting*, volume 1497 of *Lecture Notes in Computer Science*, pages 52–59. Springer-Verlag, 1998.

[RST02] Ralf Reussner, Peter Sanders, and Jesper Larsson Träff. SKaMPI: A comprehensive benchmark for public benchmarking of MPI. *Scientific Programming*, 10(1):55–65, 2002.

[RWK95] Sanjay Ranka, Jhy-Chun Wang, and Manoj Kumar. Irregular personalized communication on distributed memory machines. *Journal of Parallel and Distributed Computing*, 25:58–71, 1995.

[San02] Eunice E. Santos. Optimal and efficient algorithms for summing and prefix summing on parallel machines. *Journal of Parallel and Distributed Computing*, 62(4):517–543, 2002.

[Sch97] Robert Schreiber. High Performance Fortran, version 2. *Parallel Processing Letters*, 7(4):437–449, 1997.

[SGDM94] V. S. Sunderam, G. A. Geist, J. J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20(4):531–546, 1994.

[SKK+06] Victor Samofalov, Victor Krukov, Bob Kuhn, Sergey Zheltov, Alexandr Konovalov, and Jayant DeSouza. Automated correctness analysis of MPI programs with intel message checker. In *Parallel Computing: Current & Future Issues of High-End Computing (ParCo 2005)*, volume 33 of *NIC Series*, pages 901–908. John von Neuman Institute for Computing, Central Institute for Applied Mathematics, Forschungszentrum Jülich, 2006.

[SL04] Jeffrey M. Squyres and Andrew Lumsdaine. The component architecture of open MPI: Enabling third-party collective algorithms. In Vladimir Getov and Thilo Kielmann, editors, *18th ACM International Conference on Supercomputing (ICS), Workshop on Component Models and Systems for Grid Applications*, pages 167–185. Springer-Verlag, 2004.

[SMSY02] Hitoshi Sakagami, Hitoshi Murai, Yoshiki Seo, and Mitsuo Yokokawa. 14.9 Tflops three-dimensional fluid simulation for fusion science with HPF on the Earth Simulator. In *Supercomputing*, 2002.

[SOHL+98] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI – The Complete Reference*, volume 1, The MPI Core. MIT Press, second edition, 1998.

[SS01] Hongzhang Shan and Jaswinder Pal Singh. A comparison of MPI, SHMEM and cache-coherent shared address space programming models on a tightly-coupled multiprocessors. *International Journal of Parallel Programming*, 29(2):283–318, 2001.

[SSD+94] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The design and evolution of zipcode. *Parallel Computing*, 20(4):565–596, 1994.

[SSOB02] Hongzhang Shan, Jaswinder Pal Singh, Leonid Oliker, and Rupak Biswas. A comparison of three programming models for adaptive applications on the Origin2000. *Journal of Parallel and Distributed Computing*, 62(2):241–266, 2002.

[SSOB03] Hongzhang Shan, Jaswinder Pal Singh, Leonid Oliker, and Rupak Biswas. Message passing and shared address space parallelism on an SMP cluster. *Parallel Computing*, 29:167–186, 2003.

[SSvP07] Vijay A. Saraswat, Vivek Sarkar, and Christoph von Praun. X10: Concurrent programming for modern architectures. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, page 271, 2007.

[ST97] Horst Simon and Shanh-Hu Teng. How good is recursive bisection. *SIAM Journal on Scientific Computing*, 18(5):1436–1445, 1997.

[STF+02] Satoru Shingu, Hiroshi Takahara, Hiromitsu Fuchigami, Masayuki Yamada, Yoshinori Tsuda, Wataru Ohfuchi, Yuji Sasaki, Kazuo Kobayashi, Takashi Hagiwara, Shinichi Habata, Mitsuo Yokokawa, Hiroyuki Itoh, and Kiyoshi Otsuka. A 26.58 Tflops global atmospheric simulation with the spectral transform method on the Earth Simulator. In *Supercomputing*, 2002.

[Str04] Marc-Oliver Straub. MPI-2 one-sided communication on NEC SX-6 and IBM RS/6000 SP Power3-II. Master's thesis, University of Karlsruhe, 2004.

[Sun90] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.

[SvdVL99] Steve Sistare, Rolf van de Vaart, and Eugene Loh. Optimization of MPI collectives on clusters of large-scale SMPs. In *Supercomputing*, 1999. `http://www.supercomp.org/sc99/proceedings/techpap.htm#mpi`.

[SWP04] Gopalakrishnan Santhanaraman, Dhabaleswar Wu, and Dhabaleswar K. Panda. Zero-copy MPI derived datatype communication over InfiniBand. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 47–56. Springer-Verlag, 2004.

[TG03] Rajeev Thakur and William D. Gropp. Improving the performance of collective operations in MPICH. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 10th European PVM/MPI Users' Group Meeting*, volume 2840 of *Lecture Notes in Computer Science*, pages 257–267. Springer-Verlag, 2003.

[TG07] Rajeev Thakur and William Gropp. Open issues in MPI implementation. In *Advances in Computer Systems Architecture, 12th Asia-Pacific Conference (ACSAC)*, volume 4697 of *Lecture Notes in Computer Science*, pages 327–338. Springer-Verlag, 2007.

[TGL98] Rajeev Thakur, William Gropp, and Ewing Lusk. A case for using MPI's derived datatypes to improve I/O performance. In *Supercomputing 98: High Performance Networking and Computing*. ACM/IEEE Press, 1998.

[TGL99a] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *7th Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189, 1999.

[TGL99b] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *6th Workshop on I/O in Parallel and Distributed Systems (IOPADS)*, pages 23–32, 1999.

[TGL02] Rajeev Thakur, William Gropp, and Ewing Lusk. Optimizing noncontiguous accesses in MPI-IO. *Parallel Computing*, 28:83–105, 2002.

[TGR04] Rajeev Thakur, William D. Gropp, and Rolf Rabenseifner. Improving the performance of collective operations in MPICH. *International Journal on High Performance Computing Applications*, 19:49–66, 2004.

[TGT04] Rajeev Thakur, William D. Gropp, and Brian Toonen. Minimizing synchronization overhead in the implementation of MPI one-sided communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 57–67. Springer-Verlag, 2004.

[TNP03] Vinod Tipparaju, Jarek Nieplocha, and Dhabaleswar K. Panda. Fast collective operations using shared and remote memory access protocols on clusters. In *17th International Parallel and Distributed Processing Symposium (IPDPS03)*, page 84, 2003.

[TSY00] Hong Tang, Kai Shen, and Tao Yang. Program transformation and runtime support for threaded MPI execution on shared-memory machines. *ACM Transactions on Programming Languages and Systems*, 22(4):673–700, 2000.

[TY01] Hong Tang and Tao Yang. Optimizing threaded MPI execution on SMP clusters. In *International Conference on Supercomputing (ICS)*, pages 381–392, 2001.

[Val88] Leslie G. Valiant. Optimally universal parallel computers. *Philosophical Transactions of the Royal Society of London*, A326:373–376, 1988.

[Val89] Leslie G. Valiant. Bulk-synchronous parallel computers. In M. Reeve and S. Ericsson Zenith, editors, *Parallel Processing and Artificial Intelligence*, chapter 2, pages 15–22. Wiley, 1989.

[Val90a] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[Val90b] Leslie G. Valiant. General purpose parallel architectures. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 18, pages 943–972. Elsevier, 1990.

[vdG94] Robert van de Geijn. On global combine operations. *Journal of Parallel and Distributed Computing*, 22:324–328, 1994.

[VdS00] Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Supercomputing (SC)*, 2000. `http://www.sc2000.org/proceedings/techpapr/index.htm#01`.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus E. Schauser. Active messages: A mechanism for integrated communication and computation. In *19th Annual International Symposium on Computer Architecture (ISCA)*, pages 256–266, 1992.

[VM03] Jeffrey S. Vetter and Frank Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. *Journal of Parallel and Distributed Computing*, 63(9):853–865, 2003.

[Wal94] David W. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20(4):657–673, 1994.

[WGR02] Joachim Worringen, Andreas Gäer, and Frank Reker. Exploiting transparent remote memory access for non-contiguous- and one-sided-communication. In *Workshop on Communication Architecture for Clusters (CAC), 16th International Parallel and Distributed Processing Symposium (IPDPS02)*, page 163, 2002.

[WWP04] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. High performance implementation of MPI derived datatype communication over InfiniBand. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, page 14, 2004.

[YCM06] Hao Yu, I-Hsin Chung, and José E. Moreira. Topology mapping for Blue Gene/L supercomputer. In *ACM/IEEE Supercomputing*, page 116, 2006.

[YHG+07] K. Yelick, P. Hilfinger, S. Graham, D. Bonachea, J. Su, A. Kamil, P. Colella K. Datta, and T. Wen. Parallel languages and compilers: Perspective from the titanium experience. *International Journal of High Performance Computing Applications*, 21:266–290, 2007.

[YHK+99] Mitsuo Yokokawa, Shinichi Habata, Shinichi Kawai, Hiroyuki Ito, Keiji Tani, and Hajime Miyoshi. Basic design of the Earth Simulator. In *High Performance Computing. (ISHPC'99)*, volume 1615 of *Lecture Notes in Computer Science*, pages 269–280. Springer-Verlag, 1999.

[YIU+02] Mitsuo Yokokawa, Keníchi Itakura, Atsuya Uno, Takashi Ishihara, and Yukio Kaneda. 16.4-TFLOPS direct numerical simulation of turbulence by a Fourier spectral method on the Earth Simulator. In *Supercomputing*, 2002.

[YSP+98] Katherine A. Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul N. Hilfinger, Susan L. Graham, David Gay, Phillip Colella, and Alexander Aiken. Titanium: A high-performance java dialect. *Concurrency – Practice and Experience*, 10(11–13):825–836, 1998.

[ZS04] H. P. Zima and M. Shimasaki. Special issue: The Earth Simulator. *Parallel Computing*, 30(12):1277–1343, 2004.

Index

Part III

Collected Papers

This part constitutes the technical content of the dissertation, and consists of the 26 published papers discussed in Part II. The papers should be consulted as published and are not reprinted here; thus, there has not been any attempt to correct typos and inaccuracies.

Most of the papers are easily accessible electronically and can also be found in any well-stocked library. Pointers are given at `www.traff-industries.de/pages/disputats.html`.

1. Maciej Gołebiewski, Hubert Ritzdorf, Jesper Larsson Träff, and Falk Zimmermann. The MPI/SX implementation of MPI for NEC's SX-6 and other NEC platforms. *NEC Research & Development*, 44(1):69–74, 2003.

2. Hubert Ritzdorf and Jesper Larsson Träff. Collective operations in NEC's high-performance MPI libraries. In *International Parallel and Distributed Processing Symposium (IPDPS 2006)*, page 100, 2006.

3. Jesper Larsson Träff, Rolf Hempel, Hubert Ritzdorf, and Falk Zimmermann. Flattening on the fly: efficient handling of MPI derived datatypes. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 6th European PVM/MPI Users' Group Meeting*, volume 1697 of *Lecture Notes in Computer Science*, pages 109–116, 1999.

4. Ralf Reussner, Jesper Larsson Träff, and Gunnar Hunzelmann. A benchmark for MPI derived datatypes. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 7th European PVM/MPI Users' Group Meeting*, volume 1908 of *Lecture Notes in Computer Science*, pages 10–17, 2000.

5. Joachim Worringen, Jesper Larsson Träff, and Hubert Ritzdorf. Fast parallel non-contiguous file access. In *Supercomputing*, 2003. `http://www.sc-conference.org/sc2003/tech_papers.php`.

6. Joachim Worringen, Jesper Larsson Träff, and Hubert Ritzdorf. Improving generic non-contiguous file access for MPI-IO. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 10th European PVM/MPI Users' Group Meeting*, volume 2840 of *Lecture Notes in Computer Science*, pages 309–318, 2003.

7. Jesper Larsson Träff, Hubert Ritzdorf, and Rolf Hempel. The implementation of MPI-2 one-sided communication for the NEC SX-5. In *Supercomputing*, 2000. `http://www.sc2000.org/proceedings/techpapr/index.htm#01`.

8. Maciej Gołebiewski and Jesper Larsson Träff. MPI-2 one-sided communications on a Giganet SMP cluster. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 8th European PVM/MPI Users' Group Meeting*, volume 2131 of *Lecture Notes in Computer Science*, pages 16–23, 2001.

9. Jesper Larsson Träff. A simple work-optimal broadcast algorithm for message passing parallel systems. In *Recent Advances in Parallel Virtual Machine and Message Passing*

Interface. 11th European PVM/MPI Users' Group Meeting, volume 3241 of *Lecture Notes in Computer Science*, pages 173–180, 2004.

10. Jesper Larsson Träff and Andreas Ripke. Optimal broadcast for fully connected networks. In *High Performance Computing and Communications (HPCC'05)*, volume 3726 of *Lecture Notes in Computer Science*, pages 45–56, 2005.

11. Jesper Larsson Träff and Andreas Ripke. An optimal broadcast algorithm adapted to SMP-clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 12th European PVM/MPI Users' Group Meeting*, volume 3666 of *Lecture Notes in Computer Science*, pages 48–56, 2005.

12. Rolf Rabenseifner and Jesper Larsson Träff. More efficient reduction algorithms for message-passing parallel systems. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 36–46, 2004.

13. Jesper Larsson Träff. An improved algorithm for (non-commutative) reduce-scatter with an application. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 12th European PVM/MPI Users' Group Meeting*, volume 3666 of *Lecture Notes in Computer Science*, pages 130–138, 2005.

14. Peter Sanders, Jochen Speck, and Jesper Larsson Träff. Full bandwidth broadcast, reduction and scan with only two trees. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 14th European PVM/MPI Users' Group Meeting*, volume 4757 of *Lecture Notes in Computer Science*, pages 17–26. Springer, 2007.

15. Peter Sanders and Jesper Larsson Träff. Parallel prefix (scan) algorithms for MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI Users' Group Meeting*, volume 4192 of *Lecture Notes in Computer Science*, pages 49–75. Springer, 2006.

16. Jesper Larsson Träff. Hierarchical gather/scatter algorithms with graceful degradation. In *International Parallel and Distributed Processing Symposium (IPDPS 2004)*, page 80, 2004.

17. Jesper Larsson Träff. Efficient allgather for regular SMP-clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI Users' Group Meeting*, volume 4192 of *Lecture Notes in Computer Science*, pages 58–65. Springer, 2006.

18. Jesper Larsson Träff. Improved MPI all-to-all communication on a Giganet SMP cluster. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 9th European PVM/MPI Users' Group Meeting*, volume 2474 of *Lecture Notes in Computer Science*, pages 392–400, 2002.

19. Peter Sanders and Jesper Larsson Träff. The hierarchical factor algorithm for all-to-all communication. In *Euro-Par 2002 Parallel Processing*, volume 2400 of *Lecture Notes in Computer Science*, pages 799–803, 2002.

20. Jesper Larsson Träff. Implementing the MPI process topology mechanism. In *Supercomputing*, 2002. `http://www.sc-2002.org/paperpdfs/pap.pap122.pdf`.

21. Jesper Larsson Träff. Direct graph k-partitioning with a Kernighan-Lin like heuristic. *Operations Research Letters*, 34(6):621–629, 2006.

22. Jesper Larsson Träff. SMP-aware message passing programming. In *Eigth International Workshop on High-level Parallel Programming Models and Supportive Environments (HIPS03), International Parallel and Distributed Processing Symposium (IPDPS 2003)*, pages 56–65, 2003.

23. Guntram Berti and Jesper Larsson Träff. What MPI could (and cannot) do for mesh-partitioning on non-homogeneous networks. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI Users' Group Meeting*, volume 4192 of *Lecture Notes in Computer Science*, pages 293–302. Springer, 2006.

24. Jesper Larsson Träff and Joachim Worringen. Verifying collective MPI calls. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 18–27, 2004.

25. Jesper Larsson Träff and Joachim Worringen. The MPI/SX collectives verification library. In *Parallel Computing: Current & Future Issues of High-End Computing (ParCo 2005)*, volume 33 of *NIC Series*, pages 909–916. John von Neuman Institute for Computing, Central Institute for Applied Mathematics, Forschungszentrum Jülich, 2006.

26. Jesper Larsson Träff, William Gropp, and Rajeev Thakur. Self-consistent MPI performance requirements. *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 14th European PVM/MPI Users' Group Meeting*, volume 4757 of *Lecture Notes in Computer Science*, pages 36–45. Springer, 2007.

The conference contributions [10], [11], [14] and [26] have meanwhile (in 2008) appeared or been accepted to appear (in 2009) in journal form. These papers should thus be considered superseded by:

- Jesper Larsson Träff and Andreas Ripke. Optimal broadcast for fully connected processor-node networks. *Journal of Parallel and Distributed Computing*, 68(7):887–901, 2008. Supersedes [10] and [11].

- Peter Sanders, Jochen Speck, and Jesper Larsson Träff. Two-Tree Algorithms for Full Bandwidth Broadcast, Reduction and Scan. *Parallel Computing*, to appear 2009. Supersedes [14].

- Jesper Larsson Träff, William D. Gropp, and Rajeev Thakur. Self-consistent MPI Performance Guidelines. *IEEE Transactions on Parallel and Distributed Systems*, to appear 2009/2010. Supersedes [26].