Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank my supervisor Alexander Tiskin for his support and guidance during my work. Thanks also to the staff at the Centre for Scientific Computing, who provided access to the parallel machines used for the experiments. Furthermore, I am grateful to everyone else who helped me with technical discussions and comments on language and inconsistencies.

Declarations

I hereby declare that this thesis represents my own work and to the best of my knowledge it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any other degree at The University of Warwick or any other educational institution, except where the acknowledgment is made in the thesis. Any contribution made to the research by others, with whom I have worked at The University of Warwick or elsewhere, is explicitly acknowledged in the thesis.

Parts of the thesis have been submitted for publication [50, 51].

Abstract

The model of bulk-synchronous parallel computation (BSP) helps to implement portable general purpose algorithms while maintaining predictable performance on different parallel computers. In the last few years, frameworks for implementing BSP algorithms were proposed, each having different optimizations and programming models. This work gives an overview of approaches to implementing BSP algorithms in C/C++ or Fortran and of methods for predicting their performance. Experiments were run on three parallel machines, using optimized special purpose communications libraries for BSP algorithms. In the first set of experiments, communication and computational performance of all three parallel computers was measured separately to obtain the machine dependent parameters that describe them in the BSP model. The second and third set of experiments were concerned with measuring the performance for two common types of algorithms: memory efficient matrix multiplication and longest common subsequence computation. Based on the experimental results, we compare the performance of the matrix multiplication implementation to an optimized standard library and study performance predictability. Simple extensions to the standard BSP model for performance prediction are shown, their accuracy is evaluated and effects that cause prediction errors are discussed. The results indicate that the performance of BSP algorithm implementations can be highly dependent on the communication library that is used and hence compare their performance using different optimized communication libraries on different systems. When using the best suited library on each system, BSP implementations can achieve predictable performance and efficiency competitive with optimized standard libraries.

Abbreviations

BSP : Bulk-synchronous parallel(-ism)

BSMP : Bulk-synchronous message passing

Char-op : Character operation

CPU : Central processing unit

DRMA : Direct remote memory access

Flop : Floating point operation

LCS : Longest common subsequence (of two strings)

LLCS : Length of the longest common subsequence

MPI : Message-passing interface

Oxtool : The Oxford BSP Toolset

PUB : Paderborn University BSP-Library

PVM : Parallel virtual machine

Shmem : Shared memory

SMP : Symmetric multiprocessing

SPMD : Single program multiple data

Chapter 1

Introduction

Parallel computing has become the basis for a large part of the research in scientific computing and other sciences, particularly for applications that require large amounts of data to be processed efficiently. As high performance parallel computing resources are still quite expensive and usually shared between many users by means of job queuing systems, it is essential to have a framework for parallel programming that allows the creation of efficient algorithm implementations, independent of specific communication networks or parallel architectures. Other important issues are predictability of the running time and ease of debugging and profiling programs.

There are different theoretical models which claim to achieve all these goals, the most established being bulk synchronous parallelism (BSP) and various BSP-based models that were refined for different purposes. BSP allows simple theoretical cost modeling of parallel algorithms, independently of the underlying communication network or architecture of a particular parallel computer. As a model for parallel programming, BSP aims to achieve scalability and portability between different parallel architectures. The structure of BSP algorithms makes it easier to avoid deadlocks and enables the user to assess communication and computational costs separately, which is useful for performance profiling and performance prediction.

In the last few years, several special purpose communication libraries for BSP-style algorithms were implemented, each having a different set of functionality and optimizations, as well as debugging and profiling tools. As will be shown, the actual performance of a BSP

algorithm's implementation is very dependent on the underlying BSP communications library and its optimizations.

The aim of this study is to compare different libraries for practical BSP programming in C/C++ or FORTRAN, evaluate their performance on different parallel computers, and compare the running times to theoretical results. Therefore, two very common types of algorithms were implemented and studied under different conditions and on different parallel machines. A framework for comparing BSP libraries and a portable BSP programming library was created in C++. This library, despite its rather 'naive' implementation on top of MPI, shows better performance than existing libraries under various conditions. On top of this framework, a general purpose library with functionality for handling one- and two-dimensional data was created as a basis for matrix and string algorithms.

The remainder of this thesis is structured as follows. In Chapter 2, BSP and related models of parallel computation are introduced briefly. In particular, Section 2.3 contains an overview of approaches to practical BSP programming, and Section 2.6 describes the programming framework that was used to conduct the experiments for this work. Methods of BSP benchmarking and a discussion of the communication performance on the parallel machines that were the basis for all experiments are presented in Chapter 3. The remaining chapters show results of realistic benchmarks using algorithms for dense matrix-matrix multiplication (Chapter 4) and computation of the length of the longest common subsequence of two strings (Chapter 5). The predictability is studied using different libraries and BSP based performance models. Finally, the results are summarized and an outlook on possible future work is given in Chapter 6. The appendix contains the documentation of the software that was created, the full set of experimental results and the bibliography.

Chapter 2

Fundamentals

2.1 The BSP Model

The BSP model was introduced by Valiant in [63]. The BSP computer is defined as a set of p identical processor/memory pairs connected by an arbitrary communication network (see Figure 2.1). The original model uses the time that is needed for a simple arithmetic operation or a memory access as a base unit for computational costs and specifies the performance of the communication network relatively to this time unit. We modify this convention slightly, and include a parameter for sequential computing speed, because this thesis will be concerned with performance measurements on different processors and also for algorithms with different primitive operations (e.g. floating point add/multiply or character comparison). For simplicity, the computation speed is assumed to be constant and equal

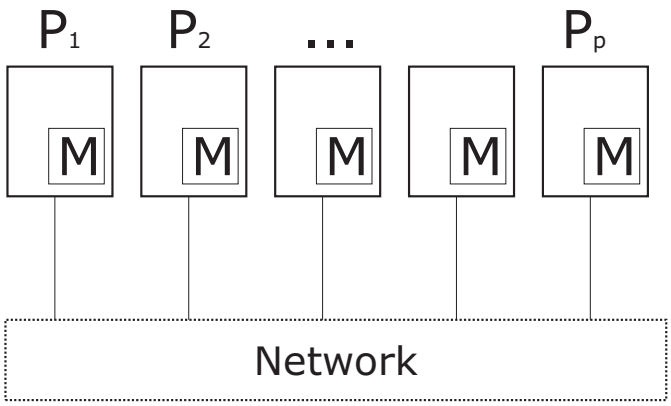


Figure 2.1: The BSP computer

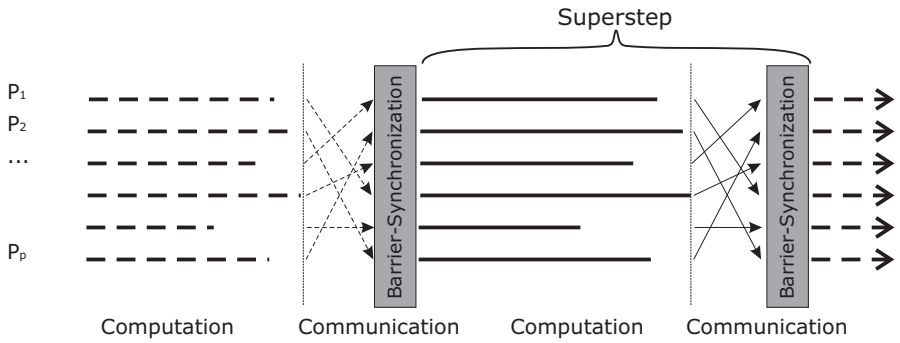


Figure 2.2: BSP program execution

for all the processors in the BSP computer. The performance of the individual processors in the parallel machine is thus characterized by the fixed time f needed to perform a primitive operation. For floating point operations, the inverse $F = 1/f$ of this parameter is equivalent to the flop rate. The performance of the communication network is characterized by a linear approximation, using parameters g and l. The first parameter, g or *communication gap*, describes how fast data can be transmitted continuously by the network after the transfer has started. Its inverse $1/g$ is equivalent to the effective bandwidth. The *communication latency* l is the time overhead that is necessary for starting up communication. Our adapted BSP model describes a BSP computer with the tuple of parameters (p, f, g, l).

Program execution takes place in single program multiple data (SPMD) style and is divided into *supersteps*, each consisting of local computations and a communications phase. At the end of each superstep, the processes are synchronized using a barrier-style synchronization (see Figure 2.2). During the computation phase, all processors perform sequential computations using the data in their local memory. In the communication phase, data is sent to the other processors. All the data that is sent during a superstep is received in the subsequent superstep. Following this scheme, it is sufficient to calculate the computational and communication costs for each superstep separately, and then the sum of these over all supersteps to get the overall cost. As communication and computation are fully decoupled, the following performance model can be used to estimate the running time.

Consider a computation consisting of S supersteps. For each specific superstep $1 \leq s \leq S$, let h_s^{in} be the maximum number of data units received and h_s^{out} the maximum number of data units sent by each processor in the communication phase. Further, let w_s be the maximum

number of operations in the local computation phase. The whole computation has separate *computation cost* W and *communication cost* H:

$$W = \sum_{s=1}^{S} w_s \tag{2.1}$$

$$H = \sum_{s=1}^{S} h_s \qquad \text{with } h_s = \max_p(h_s^{in} + h_s^{out}) \tag{2.2}$$

All the communication can be seen to take place in form of h-*relations*, i.e. collective communication operations where each processor sends and receives a maximum of $h = \max_p(h_s^{in} + h_s^{out})$ items of data to or from other processors. When implementing BSP-style communication, this can be used to optimize performance: data items that are sent to the same destination can be combined to reduce the communication overhead[1]. This also enables us to assume the communication overhead per superstep to be constant. We assume the *synchronization time* to be equal to the latency l for every superstep. The total running time is given by the sum

$$T = \sum_{s=1}^{S} T_s = f \cdot W + g \cdot H + l \cdot S \ . \tag{2.3}$$

This is sufficient as a theoretical model for BSP algorithms, and also allows simple performance prediction. However, it is worth noting that the assumption of g and l as being constant is not always true in reality. For small communication sizes h, the overhead caused by various levels of network protocols can lead to a higher effective value of g. Also, the programming model might allow the user to describe the communication in a superstep in form of messages or individual remote memory access operations. This usually involves overhead that cannot totally be avoided by implementing message combining. In the next Section, extensions to the BSP model for describing these effects will be discussed.

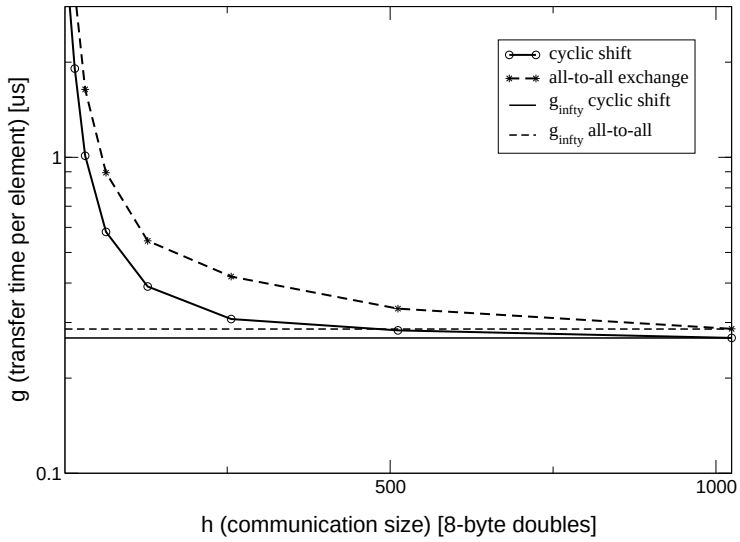


Figure 2.3: Practical values of g on a message passing system

2.2 Extensions to the BSP Model

For more realistic performance prediction in the cases where the BSP model is not sufficiently accurate, a wide range of different models and approaches exists and has been studied. A survey of BSP-like and related models is contained in [62], some of which are the Log-P/LogGP model [19, 7, 1] and the D-BSP model [20, 6]. Different BSP-like cost functions are evaluated in [5]. Blanco et al. [10] present a complex performance prediction framework, which also includes support for BSP-like algorithms. Most of these will not be considered here, as our primary aim is to preserve the simplicity of the BSP model, rather than predicting performance exactly in all possible cases. Nevertheless, there are some simple extensions to the BSP model that help to improve the accuracy of performance prediction.

The following extensions to the BSP model introduce dependencies of g on the amount of data that is sent or received. Consider one superstep with communication cost h. In the standard BSP model, h is the maximum number of data units that are sent or received

[1]Notice that the superstep structure of BSP algorithms can also be exploited to implement a more effective message scheduling, prevent contention and achieve optimal usage of the underlying communication layer. Further performance improvement can be possible on some systems by implementing optimized barrier synchronization mechanisms [39, 40].

by any processor. Even if a processor transfers less data, it still has to wait for the others, as barrier-style synchronizations take place between supersteps. When introducing more complex models for g, it can become necessary to examine whether the processor that exchanges the largest amount of data still causes the longest communication time. This can significantly complicate the performance model. A simple way of avoiding this problem is to assume balanced communication[2].

Separating Communication Startup Costs

Another view on the BSP model can be obtained by rewriting the communication time formula as follows, introducing the effective communication gap $g(h)$ as a function of the communication cost per superstep. If l_0 denotes the latency time for a barrier-style synchronization, we get:

$$\begin{aligned} T_{comm} &= g \cdot h + l \\ &= g(h) \cdot h + l_0 \ . \end{aligned} \tag{2.4}$$

This is useful to separate the actual communication startup cost for different types of communication from the actual barrier synchronization latency. It can be observed that $g(h)$ becomes larger when the communication size h in a superstep is low. This behavior shows very clearly when g is determined by measuring the time for h-relations of different sizes. Figure 2.3 shows an example for values of g (the values were measured using MPI on a 16 processor cluster system). The communication gap increases drastically for small values of h. This can be accounted for by introducing the parameters h_{half}, which is the minimum communication size where at least half of the optimal bandwidth is obtained, and the asymptotic communication gap g_∞. The effective value of g can then be defined for every superstep:

$$g(h) = \left(\frac{h_{half}}{h} + 1 \right) \cdot g_\infty \ . \tag{2.5}$$

[2]For a model of unbalanced communication and an experimental study, see [48, 49].

One problem of this approach is that cost analysis can become slightly more difficult, since not only the overall communication cost for all supersteps must be known, but also the communication cost in the individual supersteps – each superstep can now have a different effective value of g.

Including Per-Message Overhead

Another problem arises when the programming model allows the programmer to send multiple messages in one superstep, or when there are functions for remote memory access (e.g. `put`/`get` primitives). Every message or remote memory access may incur overhead, depending on the efficiency of the message combining implementation of the communication library. Different ways of modeling overhead per message are included e.g. in the LogP [19], BSP* [47] or QSM [59, 32] models. Another related extension to the BSP model is proposed and discussed e.g. in [57, 37]. The model used here is most similar to BSP*. Suppose each processor exchanges a total of h data units. We assume that these are sent in messages of size h^*. Similarly to (2.5), we make g a function of the communication cost per superstep h and message size h^* by introducing another parameter o, which specifies the overhead per message[3]:

$$g(h, h^*) = \left(\frac{h_{half}}{h} + \frac{o}{h^*} + 1 \right) \cdot g_\infty \ . \tag{2.6}$$

This approach assumes all messages in a superstep to be of equal size. However, if the message size varies, the size of the largest message[4], or the average message size can be used for h^*. This is an even greater step away from the simple BSP model, and only makes sense when the value of h^*, or at least an approximate ratio between h^* and the communication cost per superstep h, is known. When using this model, it is necessary to assume that all processors send messages of similar size. Otherwise, it can become necessary to find out precisely which processor is causing the longest communication time, because the communication time also depends on the message size. Let e.g. $h_{half} = 500$ bytes and

[3] This parameter is equivalent to the parameter $N_{\frac{1}{2}}$ from [57, 37], which denotes the message size for which half of the asymptotic bandwidth is obtained.

[4] This is sensible because the communication time will presumably be dominated by the time needed for transferring messages of this size – small messages cause a higher value of g, but also require less data to be transferred.

$o = 200$ bytes. If one processor sends 1000 bytes of data using one message, the transfer time can be calculated as

$$g(1000, 1000) \cdot 1000 = 1700 \cdot g_\infty \ .$$

If another processor sends 500 bytes in five individual messages, this will cause a longer transfer time:

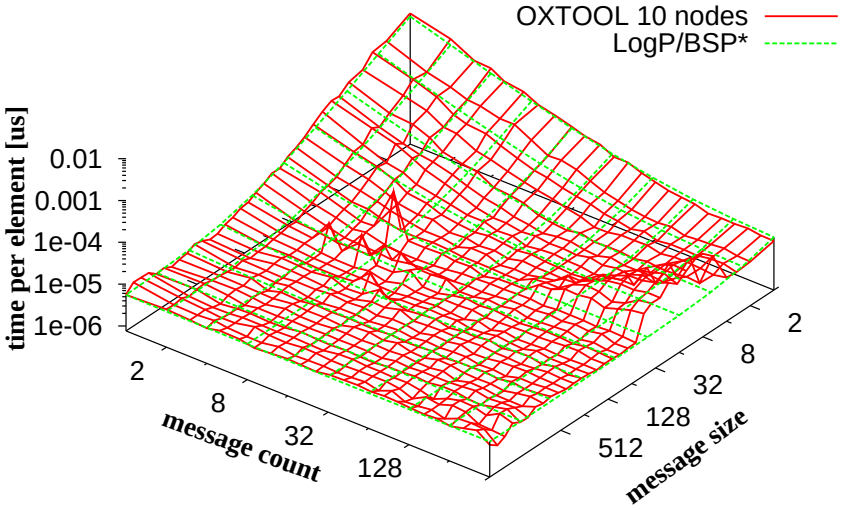
$$g(500, 100) \cdot 500 = 2000 \cdot g_\infty \ .$$

Hence, for using this approach, the communication must be balanced and consist of messages of similar size.

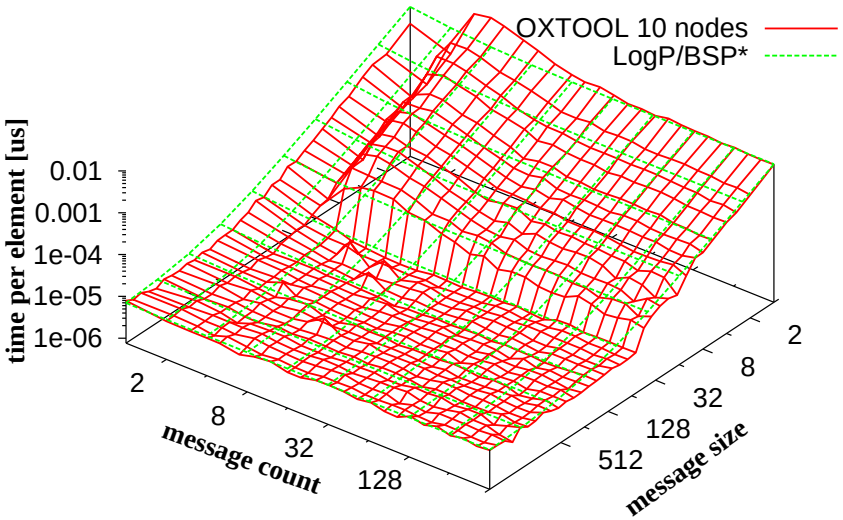
The additional parameters h_{half} and o can also be used to compare BSP communication libraries. The parameter h_{half} can be regarded as a measure for the overhead caused by network protocols and the message scheduling facilities of a BSP library. A low value of h_{half} indicates that these facilities work efficiently, and that the time necessary for starting communication is low. High values of h_{half} show that a large amount of data needs to be transferred to make use of the maximum network bandwidth. The parameter o is a measure for the overhead caused by individual calls to communication primitives. When o is very low for a certain communication primitive, it does not matter whether a fixed amount of data is transferred using many 'small' calls or one 'large' call of the primitive.

Figure 2.4 shows values of g measured[5] for two different communication primitives, and also an approximation using (2.6) (notice that the scale is logarithmic on all axes). These values were obtained by using the remote memory access primitives `put` and `get` to perform a random permutation and taking communication time samples. Each point on the graph surface corresponds to the effective value of g for a superstep with communication cost h, where every processors sends messages of fixed size to exactly one other processor. The *message size* axis corresponds to h^*, the *message count* axis indicates the number of messages $\frac{h}{h^*}$. For the `put` requests, the value of o is low. This reduces the contribution of the $\frac{o}{h^*}$ term in Equation (2.6), making the model similar to Equation (2.5). `Put`-like communication can

[5] The values were sampled on an Ethernet based PC cluster that will be introduced as `argus` in Section 2.4. The spikes that can be seen in Figure 2.4(a) occurred randomly and are presumably caused by the communication network being more or less busy.



(a) Put



(b) Get

Figure 2.4: Bandwidth gap for different communication primitives

be optimized by combining `puts` to the same destination. Therefore, many `put` requests can be carried out more efficiently on this system and have virtually no per-message overhead. For executing `get` requests on an Ethernet-based message passing system, it is necessary to request the data from the source node before the actual data can be transferred. This causes an overhead per request and leads to longer transfer times per data element (i.e. higher values of g) when there are many requests. This type of behavior can be modeled more accurately using Equation (2.6).

2.3 BSP Programming

There are various approaches to implementing BSP algorithms, each having their specific drawbacks and advantages. For the scope of this work, only communication libraries for C/C++/Fortran that support BSP-style communication were taken into consideration. Since most communication libraries are in principle usable for implementing BSP algorithms, special attention is directed to libraries that implement optimization methods for BSP algorithms.

A first approach to BSP programming in C/C++ or Fortran might be to use general purpose communication libraries like MPI [61, 34] or PVM [27], and write programs in 'BSP style'. An example is given by Bisseling in [9] and on the corresponding website [8] as project MPIedupack.

Another approach is to use an implementation of the BSPlib library interface for direct BSP programming [38, 60]. It was introduced to simplify BSP programming and keep platform independence by having a common standard. It includes the main primitives for synchronization, remote memory access and bulk synchronous message passing. Furthermore, it also allows the implementation to have more advanced optimization methods, like optimized barrier synchronization, combining messages to the same destination [39] and randomized routing.

Two implementations of the BSPlib standard for C/C++ and Fortran programmers exist. One of them is the Oxford BSP Toolset (Oxtool) [76]. It has been optimized separately for message passing, native direct remote memory access (DRMA) and shared memory platforms – the latter is particularly useful as on some shared memory/DRMA platforms, emulated

message passing can lead to additional overhead. Oxtool implements optimized message scheduling controlled by static parameters that have to be determined for each system it is run on. The distribution package comes with various tools for call-graph profiling and performance analysis. It also includes support for checkpointing and process migration on some systems.

The other implementation of BSPlib is the PUB library from Paderborn University [75, 12]. PUB includes some features not present in the BSPlib standard, such as oblivious synchronization[6] and partitioning processors into subgroups. Both these features can lead to better synchronization performance. The authors of PUB argue that the BSPlib standard is not sufficient to provide optimal performance for certain collective communication operations. Consequently, PUB provides optimized communication functionality for broadcasting and reducing data. Like Oxtool, PUB also includes a graphical tool for profiling.

Another related project is CGMlib [69], an implementation of CGM, a model for parallel programming similar to BSP. CGMlib runs on top of MPI and provides a high level programming interface in C++ for communicating lists of abstract datatypes of constant size. This differs from the BSPlib approach, where messages contain arrays of bytes that can have variable length. CGMlib also provides implementations of advanced communication primitives (broadcast, h-relation). There are implementations of various standard algorithms like sorting and parallel prefix, as well as of several CGM graph algorithms [13] for this library.

Finally, there is a project called SSCRAP (Soft Synchronized Computing in Rounds for Adequate Parallelization) [24], which provides a C++ library that contains primitives for synchronization, communication and processor group management. It runs on top of MPI or Posix Threads. The authors of SSCRAP propose a 'soft' synchronization mechanism. They argue that the latency can be reduced by not performing a full barrier synchronization, and provide different kinds of synchronization that only require individual processors to wait until either all of their outgoing messages have been sent or all incoming messages were received. Algorithm implementations for list ranking, sorting and prefix sums are available.

A distinction between these libraries can be made by comparing the communication prim-

[6] Oblivious synchronization requires the number of messages that are expected to be received on each processor to be known locally and passed to the synchronization function. In general, the processors would need to exchange this information first.

itives and advanced features they provide. Simple communication primitives are direct synchronous/bulk synchronous message passing (MP/BSMP) and DRMA. A feature that can improve performance is processor group partitioning and subgroup synchronization. PUB and Oxtool include support for process migration; this can be useful when using a BSP library on a network of workstations. The C++ libraries SSCRAP and CGMlib introduce the feature of an abstract h-relation, allowing the programmer to describe BSP-style data exchange operations for abstract data types. A brief comparison between some of these features is shown in Table 2.1; another list can be found in [24]. Bisseling [9] also gives a summary of different BSP programming environments and shows benchmark results for various parallel computers.

Feature	**Oxtool**	**PUB**	**CGMlib**	**SSCRAP**	**MPI**
MP	-	-	•	•	•
BSMP	•	•	-	[a]	[b]
DRMA	•	•	-	•	[c]
Broadcast	•	•	•	•	•
Abstract h-relation	-	-	•	•	-
Process groups	-	•	•	•	•
Process migration	•	•	-	-	-

[a] MPI-like non blocking send and blocking receive
[b] Can be implemented using `MPI_Isend`
[c] Only available in MPI-2

Table 2.1: Feature comparison between BSP libraries

Oxtool and PUB have been chosen for our experiments because they implement advanced optimizations to reduce latency and improve the effective bandwidth. CGMlib and SSCRAP rely more heavily on optimizations in the underlying communication layer (MPI) and also have a different programming library interface, thus they can not easily be compared to PUB and Oxtool using a BSPlib style programming model. A C++ wrapper library, called BSPWrapper, was created. It provides a front end to BSMP, DRMA, broadcasting and reduction operations in order to compare the performance of different libraries within the same algorithm implementations. The documentation of Oxtool includes a warning that its MPI implementation has rather high latency and only suboptimal performance; however, it is still used for our experiments, because it is portable to all of the target machines. In particular, two of the machines used for the experiments have interconnection networks that

are not directly supported by any of Oxtool's devices. For the same reason and to ensure comparability, PUB was also compiled on top of MPI. The only exception was the shared memory machine, on which modified version of PUB's shared memory communication interface was used. On the Ethernet system, the only reason for using MPI was the ability to run the experiments under control of the queuing system. If this were not necessary, Oxtool and PUB could have both been compiled using their TCP/UDP-IP communication device to improve performance. On the other hand, this would involve further effort for setting up the libraries. Further, on many machines that are intended for running production code, MPI is the only alternative when an installation of Oxtool or PUB is not available. Therefore, using MPI on an Ethernet system may well reflect the typical usage scenario. Finally, a pure MPI implementation that provides BSPlib style functionality using non-blocking send/blocking receive operations was created to compare the performance of PUB and Oxtool to an unoptimized library.

2.4 Experimental Environment

All of the experiments were conducted on three different parallel machines at the Centre for Scientific Computing at the University of Warwick. The first one (`skua`) is an SGI Altix shared memory machine with 56 Itanium-2 1.6GHz nodes and a total of 112GB main memory. The second one (`argus`) is a Linux cluster with 31 $\times$ 2-way SMP Pentium4 Xeon 2.6 GHz processors and 62 GB of memory (2 GB per 2-processor SMP node); it has good local computation performance but a slow communication network (100 Mbit Ethernet). The third system (`aracari`) is an IBM Myrinet cluster with 64 $\times$ 2-way SMP Pentium3 1.4 GHz processors and 128 GB of memory (2 GB per 2-processor SMP node). It offers very good communication performance, but has slow individual processors. The experiments used up to 32 processors on `skua` and `aracari`. The tests on `argus` had to be restricted to a maximum of 10 processors, in line with the generally expected usage of this system. As all three systems are machines used for running production code and require computation time to be requested using a job queuing system, it could not always be guaranteed to get the same CPUs for different jobs. Furthermore, the queuing system limits the number

of processors that are usable for experiments, as jobs requesting too many CPUs stay in the queue and cannot be executed. Functionality to request nodes connected in a certain network structure was intentionally not used. Furthermore, the traffic on the communication network was different every time because other users were running jobs on the remaining nodes. However, these conditions provide a realistic testing environment and can show well, how differently optimized communication libraries perform. Table 2.2 shows the low-level interface used by the libraries on each system.

2.5 Implementation Specific Observations

The Oxford BSP Toolset

The Oxford BSP Toolset was compiled on top of its message passing MPI device. A few adaptations to its compilation scripts had to be made for it to compile on the SGI Altix system using the Intel C compiler and the SGI's shared memory MPI library, and also to link with the mpich libraries used on the other systems. Changes were made to `bspfront.lprl`, `bspfrontenv.lprl` and `bsparch.lprl`. Depending on the compiler that was used, further modifications had to be made to include correct files at certain points. Considering that the library was last updated in 1998, relatively few changes were necessary. However, a serious problem for future systems could be its restriction to 128 processors (although the configuration script allows 1024 processors internally) when using MPI. It is very important to update the file `<Oxtool root>/include/bsp_parameters.ascii` with correct values for bandwidth, computational speed and latency. Otherwise, default parameters are used and only suboptimal performance can be achieved. The Perl script `parse_bspprobe_data.pl` (see Appendix B.2) can be used to extract these parameters from BSPWrapper's `bspprobe` results. Together with `mpich-gm` on Myrinet [72], certain communication patterns were

	Oxtool	**PUB**	**MPI Library**
`skua`	MPI-1	shmem	MPI-2
`argus`	MPI-1	MPI-1	MPI-1
`aracari`	MPI-1	MPI-1	MPI-1

Table 2.2: Communication interface used on different systems

observed to create conditions that can cause a segmentation fault related to the call to `MPI_Isend`. A workaround for this is to increase the BSP FIFO size (for the benchmark and matrix multiplication experiments, `bspfront -bspfifo 262144` was used, the experiments from Chapter 5 were run using `bspfront -bspfifo 512`). On some MPI systems, it is necessary to call `bsp_init_setup` before `bsp_begin`, passing the actual values of `argc` and `argv`. Otherwise a segmentation fault occurs when calling `bsp_begin`.

PUB

On the distributed memory systems, PUB was compiled using its MPI communication device. On Myrinet there were some stability issues that could only be avoided by picking a send buffer size that 'works'. It seemed that the `mpich-gm` installation used for the experiments together with PUB could cause data corruption in PUB's message buffers. This problem did not occur on the Ethernet system, which used `mpich` version 1.2.5 [71]. The buffer size can be adjusted by compiling `bspwrapper_pub.cpp` with `-DPUB_SENDBUFSIZE=xxx` to initialize PUB with the updated value. On the Myrinet system, a clear drop in communication performance could be observed for messages of a certain size. This occurs when PUB switches the sending method for messages too large to fit into the send buffer. For running on our shared memory machine, the shared memory device of PUB was modified by adding an adapted version of the Cray/Unicos device and making minor modifications to `cl/shmem_common.c|h` to replace the `enum` datatype used for their locking mechanism with integers. Also, a deadlock condition in the packet transfer mechanism was removed after correspondence with the PUB authors. The changes can be found at
`http://www.warwick.ac.uk/staff/P.Krusche/bsplibraries.html`.

MPI Message Passing (MPIMPASS)

The message passing MPI implementation of BSPWrapper uses `Isend` and `Recv` to transfer messages. For each message, a header is transferred when `Sync` is called. The headers are transferred either by calls to `MPI_Alltoall(v)` or `Isend` depending on whether the C preprocessor definition `_ISEND_HEADERTRANSFER` is set. A simple message combining

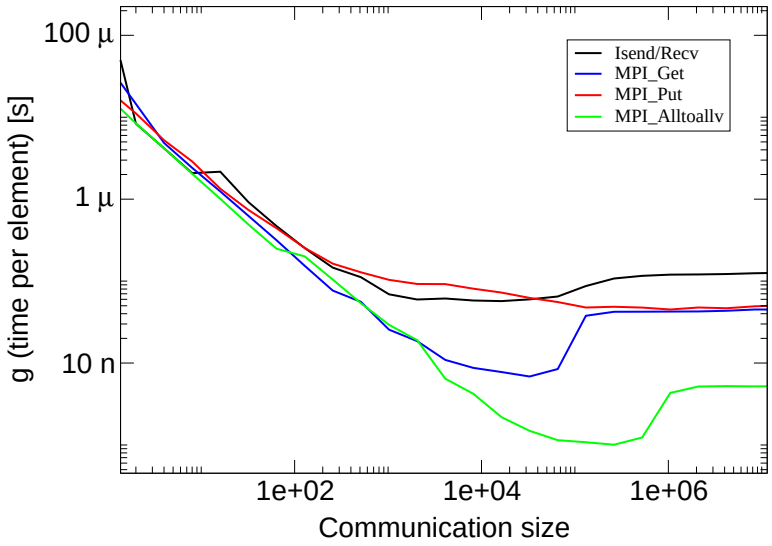


Figure 2.5: Communication gap for all-to-all exchange using different MPI primitives

implementation can be activated by defining `MPI_MAXCOMBINESIZE` as the maximum size of a message that can be combined with other messages. This does not necessarily have to be the fastest way for transmitting messages, as Figure 2.5 (sampled on `skua`) shows, but it turned out to be the one that involves the least programming effort for implementing BSPlib-style bulk-synchronous message passing. When using `mpich-p4 1.2.5` [35, 36] on `argus`, problems occurred when `mpich`'s shared memory segment size was too low. It can be specified using the environment variable `P4_GLOBMEMSIZE` [36] before running the program, but the size of this segment could not be increased beyond a limit of 32 MB. This problem can also occur on other systems using `mpich`. Solutions are to recompile `mpich` with different settings (if possible), or to reduce the communication cost per superstep. Different problems occurred when using `mpich-gm` (Version 1.2.5-10 using `gm` 1.6.4). This MPI implementation appears to have difficulties with too many `Isend` requests. Especially when using many nodes and for large communication costs, this also caused the Oxtool and PUB variants of BSPWrapper to terminate. MPIMPASS showed less sensitivity to this issue after implementing header transfer using `MPI_Alltoall(v)` instead of `MPI_Isend`.

The emulated DRMA layer of BSPWrapper uses these functions to transfer the data and

is contained in `bspwrapper_gendrma.cpp`. It buffers all `Put` and `Get` requests and carries them out upon synchronization. In the first round, headers for all `Get` requests are sent to the nodes that hold the requested data and all the `Put` data is sent. In the second round, the `Put` data is received and the `Get` requests are processed. In the third and final round, the data for the `Get` requests is received. Obviously, this mechanism is not optimal in terms of latency and overhead. Therefore, it should only be used on systems that do not provide MPI-2 DRMA primitives, otherwise the MPISHMEM implementation will provide better performance.

MPI Shared Memory (MPISHMEM)

This implementation is identical to the MPIMPASS implementation except for the DRMA layer. DRMA is implemented using the MPI-2 functions `MPI_Get` and `MPI_Put`. This has the advantage that executing the unbuffered primitives (`HpPut`/`HpGet`) can be interleaved with local computations, if the underlying implementation of MPI supports this. On our shared memory system, DRMA operations generally achieved better performance than MPI's message passing functions. No `Put` combining as in the other libraries takes place on top of MPI, but `Put` and `Get` requests are buffered to preserve the same semantics as in BSPlib.

2.6 Our Programming Framework

As discussed in the previous sections, a wrapper library has been designed for comparing BSP programming libraries. It was created to provide a uniform front end in C++ for running the same algorithm implementation on top of every library. If no optimized library is available, the library can be compiled using MPI-1 or MPI-2, depending on the version that is available on the specific platform. The library interface is based on the BSPlib standard and provides front end functions in the C++ namespace `BSPWrapper`. The reason for creating this wrapper library, rather than using BSPlib directly, is that C++ was used to create the 'naive' MPI back end libraries and also to implement the algorithms described in Chapters 4 and 5. Also, small differences between the BSPlib interfaces of Oxtool and PUB (mainly concerning library initialization) are hidden by the wrapper library. The programming interface will be

introduced now to clarify the basis for all the software that was used in the experiments. Further documentation can be found in Appendix A.

Global Variables

```
/* P contains the number of nodes */
extern int P;
/* Pid identifies our node */
extern int Pid;
```

Notice that these variables only contain valid values after the call to `Init()`.

Initialization and Exit

```
/* Initialization */
void Init(int argc, char ** argv);
void Exit();
```

`Init` initializes the BSP library. The parameters `argc` and `argv` are the command line parameters given to `main`, they are necessary because some MPI implementations are not able to initialize without them.

`Exit()` frees all buffers used by BSPWrapper and calls the library's exit function.

Synchronization

```
/* Synchronize */
void Sync();
```

The function `Sync` is responsible for performing a barrier synchronization and conducts all buffered communication.

Direct Remote Memory Access (DRMA)

```
/* Collective registration/deregistration */
void PushReg(void* data, int len);
void PopReg(void* data);

/* DRMA primitives */
void Put(int pid_dest, void * src, void * dest, int offset,
```

```
        int count);
void Get(int pid_from, void * src, int offset, void * dest,
        int count);

void HpPut(int pid_dest, void * src, void * dest, int offset,
           int count);
void HpGet(int pid_from, void * src, int offset, void * dest,
           int count);
```

The interface for DRMA is based on the BSPlib standard. Every buffer used as a destination for `(Hp)Put` or as a source for `(Hp)Get` has to be registered first with `PushReg`, which is a collective operation. After the next synchronization, data can be exchanged by using buffered (`Put` and `Get`) and unbuffered access functions (`HpPut` and `HpGet`). After remote access to the buffer is not needed anymore, it should be freed with `PopReg`.

When using buffered `Put` access, data is copied from the source buffer right after the function call, and not written to the destination until the end of the superstep. This means all the buffers involved can be reused until the call to `Sync`. The unbuffered equivalent `HpPut` requires the source buffer to remain unmodified and leaves the destination undefined until the end of the superstep, as it allows the library to start the data transfer right away. When using buffered `Get` access, the actual copying takes place at the end of the superstep. In general it can be said that, at the expense of memory and copying time, the buffered operations are safer and easier to use.

After the call to `PopReg()`, the buffer can only be freed after the next `Sync()` operation, otherwise the implementation using the PUB library became unstable in some experiments. Buffers that are not currently used for memory access should not be registered, as this may increase lookup times and overhead for the DRMA function calls depending on the implementation.

Bulk Synchronous Message Passing (BSMP)

```
/* BSMP primitives */
void Send(int pid, void* tag, void* message, int count);
void GetTag(int * status, void* tag);
void Move(void * data, int size);
void SetTagSize(int sz);
```

BSPlib-style bulk synchronous message passing (BSMP, see e.g. [60]) can be used for more

complex communication patterns that cannot be easily described by DRMA operations. Once BSMP messages are sent, they will arrive in the next superstep. The tag size has to be fixed and can be changed in one superstep with `SetTagSize`. The basic procedure for sending and receiving messages is illustrated by the following piece of code; it sends 10 bytes of data to every processor and then receives 10 bytes from every processor.

```
SetTagSize(sizeof(int));
int tag= Pid;
char data[10]
char receivebuffer[10*P]

/* send data to each processor */
for(int p= 0; p < P; ++p)
    Send(p, &tag, data, 10);

Sync();

int status= 0;
while(status >= 0) {
    GetTag(status, tag);
    if(status > 0)
        Move(receivebuffer[10*tag], status);
}
```

After the call to `GetTag`, the variable `status` contains the length of the message that was received, or -1 if no messages are left in the queue. The `Send` function buffers data and all the buffers passed to it can be reused after the call. Thus, our BSMP interface essentially has the same semantics as the BSPlib standard. However, as shown in Appendix A, the MPI version of BSPWrapper also supplies unbuffered BSMP functions, which enable the improvement of memory efficiency.

Advanced Communications

```
/*
    Broadcast one value from pid -> all
    (collective operation)
 */
void Broadcast(int pid, int count, void* src, void * dest);

/*
    Folding with operation
        fun(res, a, b, len)
    (collective operation)
```

```
 */
void Fold(void* src, void * dest, int count,
          void fun (char*, char*, char*, int*));
```

Some libraries provide predefined primitives for collective communication that have better performance compared to the implementation of this operation with BSMP/DRMA primitives. Because of this, the following functions from the second draft to the BSPlib standard were included: The function `Broadcast` copies data from `src` on node `pid` to `dest` on all nodes. The function `Fold` combines data using the operator `fun()`. The operands to fold are given by the pointer `src` on each node, the result is returned in `dest` on all nodes. The operator function `fun()` must accept four arguments. The first pointer receives the result, the next two are operands. The data pointed to by the `int` pointer specifies the size of the data that was specified by `count`.

Chapter 3

BSP Benchmarking

This Chapter describes how the parameters of a BSP computer as referred to in Chapter 2 can be determined. Further information on the practical implementation of the benchmarking and data analysis routines created for this thesis can be found in Appendices A and B.

There are various approaches to obtaining the BSP parameters on a parallel machine. Work on this problem has been done by the authors of the Oxford BSP Toolset [76], PUB [75] and by Bisseling [9]. Another related study is [29], where the performance of the remote memory access functions of various communication libraries (Oxtool, PUB and LAM-MPI [70]) is compared for implementations of several algorithms. Study [29] also presents a framework for communication library independent BSP-like programming based on DRMA `put` and `get` primitives. Juurlink and Wijshoff [49] study the predictability of various algorithms using Oxtool and different performance models. Further benchmarks and work on the validity of the BSP model are contained in [28, 67].

3.1 Measuring the Processor Speed

The computation speed of an individual processor in a BSP computer is usually determined by measuring the number of sequential operations of a certain kind it can perform per second. A very common measure for the computation speed of computers is their flop rate F, which denotes the number of floating point operations per second that can be performed by a specific processor. In the following, all 'flops' will also correspond to double precision (i.e.

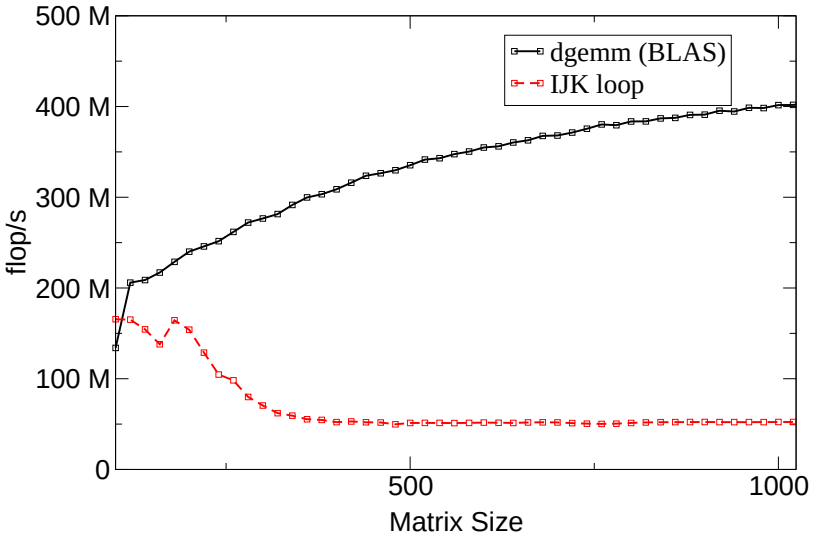


Figure 3.1: Matrix multiplication — sequential performance

64-bit) floating point operations. All message sizes will be given in units of double values. As discussed in Chapter 2, the computation speed of CPUs in a BSP computer is represented by the inverse flop rate $f = 1/F$, i.e. the time for one floating point or other arithmetic operation. Many benchmarking tools for determining the flop rate of a single processor exist. One example is LINPACK [23, 21], which is based on solving a dense system of linear equations. Another source for various benchmarks is the Standard Performance Evaluation Corporation (SPEC) [80].

Both of the optimized BSP libraries require an approximation of F for setting up their message scheduling. The distribution of the Oxford BSP Toolset therefore includes a tool called `bspprobe` that measures the average of the flop rates for a dot product and dense matrix multiplication using an IJK loop (both using input data that preferably is larger than the CPU cache size). The PUB library measures an approximation of f when the user program is started, based on measuring the time for memory copy operations. Our benchmarking tool measures f in a similar way as `bspprobe` from the Oxford BSP Toolset. However, our measurement functions use BLAS [22, 73] for floating point data types, and an adapted version of the original `bspprobe` code to measure integer performance. The values

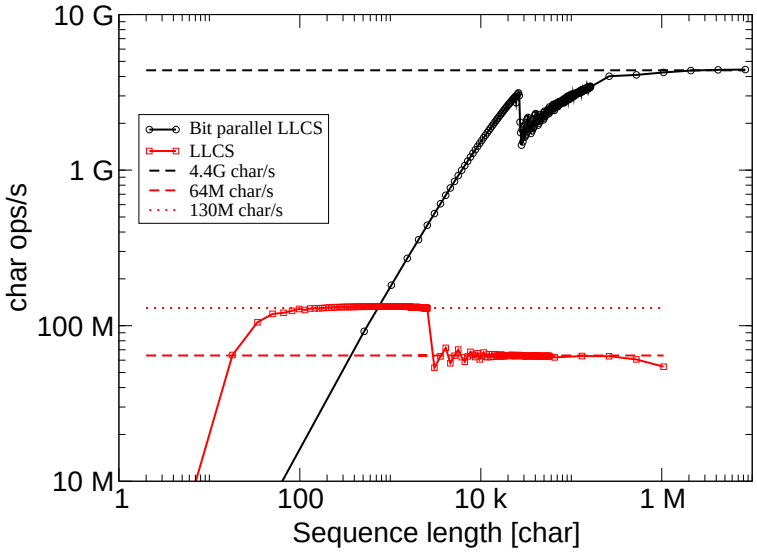


Figure 3.2: LLCS computation — sequential performance

for f are based on the time for computing a dot product and for dense matrix multiplication. Tables C.1, C.4 and C.7 in Appendix C contain the values of f that were measured on the systems introduced in Section 2.4. However, numerous problems can arise when trying to use these approximations for predicting performance:

- The running time varies for different operations and data types.
- F also depends on other factors, such as the level of the CPU cache hierarchy or main memory from which data has to be fetched.
- Algorithmic overhead reduces the effective value of F for small problem sizes. Therefore, the observed values of F and f are functions of the input size.

Figure 3.1 shows the performance of two algorithm implementations for multiplying square dense matrices. One uses a simple IJK loop, the other runs an optimized implementation of the level-3 BLAS primitive `dgemm` from ATLAS [68]. The IJK loop has a performance peak for matrices that fit into the CPU cache. For larger matrices, performance drops and becomes constant. The behavior of `dgemm` is different; it shows much better asymptotic performance, which is approached from below as the problem size increases.

In Figure 3.2, the performance of two different algorithm implementations is compared, both computing the length of the longest common subsequence of two strings. Their basic common operation is character comparison: the dynamic programming algorithm compares $O(n^2)$ character pairs, and the bit-parallel algorithm does something similar on bit-level, using a character to bit string mapping (for more information on both algorithms, see Chapter 5). The simple dynamic programming implementation shows a behavior similar to the IJK loop for matrix multiplication. It reaches maximum performance until a cache threshold point is reached, where the amount of data that is needed to be stored in the CPU cache is too large and cache efficiency decreases. The bit-parallel variant uses a more efficient algorithm proposed by Crochemore et al. [18], that stores the dynamic programming table in a more compact form and uses integer operations to compute the values of multiple entries 'in parallel' (notice that this is processor-level parallelism and does not involve any parallelism on the level of a BSP computer). It also shows a performance drop at a different cache threshold point, as it stores the data in a more compact form. However, on this particular system, performance still rises after passing the threshold point. We used C++ to implement the bit-parallel algorithm. Implementing bit-level data access and addition with carry (see Chapter 5) using a high-level programming language necessarily causes more overhead for small sequence lengths.

In conclusion, the value of f can only be considered asymptotically constant. To keep the simplicity of the BSP model, our approach of obtaining f for a specific algorithm is to measure the performance of the sequential implementation separately in one run for every parallel algorithm that is studied. For predicting the performance in Chapters 4 and 5, the computation phase is timed on one processor to obtain an asymptotic approximation of f.

3.2 Measuring the Communication Gap

The parameter g can be seen as the inverse bandwidth of the communication network between processors in the BSP machine. It is the time needed to transfer one element of data. For testing communication performance, units of 8-byte double values are transmitted, because the same data type is used for local computations in Chapter 4 (Matrix Multiplication). The

`bspprobe` tool in its original version from the Oxford BSP Toolset measures g for two kinds of communication: a cyclic shift, where each processor sends and receives data to/from exactly one other, and an all-to-all exchange, in which each processor sends data to all the others. It takes samples for different message sizes and estimates the values of g_∞, which it assumes to be the minimum value of g measured, and o (see Section 2.2 for the definitions). The value of o (named $N_{\frac{1}{2}}$ by the author of `bspprobe`) is calculated using the ratio between the maximum and minimum measured gap values g_{max}/g_{min}, and the communication cost h_{max} for which g_{max} was achieved:

$$o = \left(\frac{g_{max}}{g_{min}} - 1\right) \cdot h_{max} \ . \tag{3.1}$$

One problem with this method is that the performance can be dependent on the communication primitive that is used (cf. e.g. [29]). `Bspprobe` uses the unbuffered `hpput` primitive of the BSPlib standard. When it comes to predicting performance, the values of g_∞ and o for `get` and `send` can differ from the ones obtained using `hpput`. Inaccuracies in the parameter estimations can be caused by spikes of very long communication time, which were observed on some library and system combinations. An example of this can be seen in Figure 3.3. Particularly MPI showed such spikes apparently at random. Presumably these spikes are caused by varying communication network traffic.

A better approach for obtaining the parameters g and l is to measure communication times for different numbers of messages, and then perform a linear regression for this data (see Bisseling [9]). If communication time shows approximately linear increase with the number of messages, the slope of the fitted line gives g and the value at the intersection of this line with the time-axis, i.e. the time for '0 messages', gives l. This approach has the advantage of using data for different message sizes to minimize the statistical error when estimating g, but it also has the drawback of only taking samples for one message size and ignoring the per-message overhead o.

To get a more precise picture of how well the different communication libraries work, we measure the performance for different numbers of messages $c = \frac{h}{h^*}$ (as the communication cost in this superstep is equal to the product of the number of messages and their size:

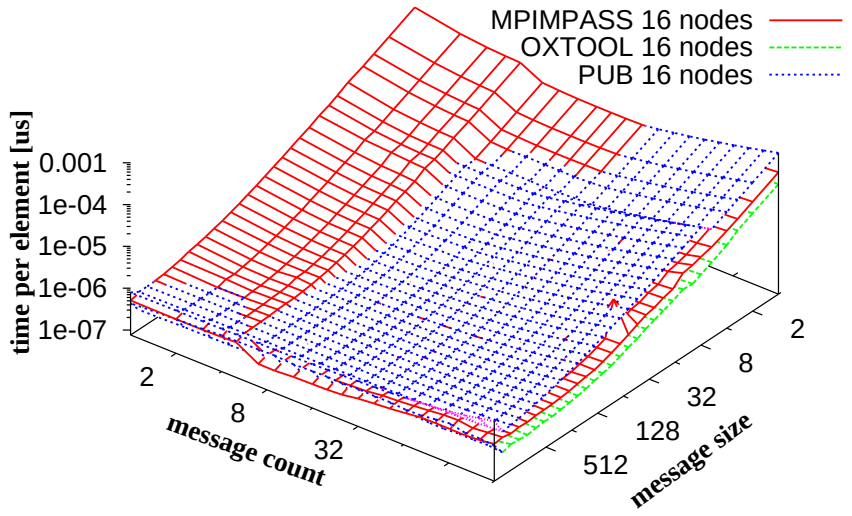


Figure 3.3: Example of communication gap surface measured by our benchmarking tool

$h = c \cdot h^*$) and message sizes h^*. The value of g is estimated by measuring the time for one superstep performing an all-to-all exchange or a random permutation, which is implemented using multiple `Put`, `Get` or `Send` requests of equal size, equivalent to the messages of size h^* from Section 2.2. The resulting data can be visualized and compared as shown in Figure 3.3. The scale of all axes is logarithmic to show more detail in the regions where g does not vary much. The value of g_∞ is determined by taking a weighted average of the sampled data $\tilde{g}(h^*, c)$:

$$g_\infty \quad = \quad \sum \tilde{g}(h^*, c) \cdot \omega / \sum \omega \ . \tag{3.2}$$

The weight function $\omega = (c \cdot h^*)^3$ was chosen to provide reliable results when the data is noisy. This is equivalent to weighing the measured running times of the superstep with the squared communication cost, as

$$\tilde{g}(h^*, c) \cdot (c \cdot h^*)^3 = \underbrace{\tilde{g}(h^*, c) \cdot h}_{T_{comm}} \cdot (\underbrace{c \cdot h^*}_{h})^2 \ .$$

We assume that the effective communication gap will attain its largest values for a low

communication cost per superstep. The value of g for low communication costs is computed in a similar way, using $\omega' = (c \cdot h^*)^{-3}$ as the weight for averaging:

$$g_{small} = \sum \tilde{g}(h^*, c) \cdot \omega' / \sum \omega' \ . \tag{3.3}$$

The value of h_{half} can be obtained using h_{min} as the lowest communication cost, for which the communication time was measured:

$$h_{half} = \left(\frac{g_{small}}{g_\infty} - 1 \right) \cdot h_{min} \ . \tag{3.4}$$

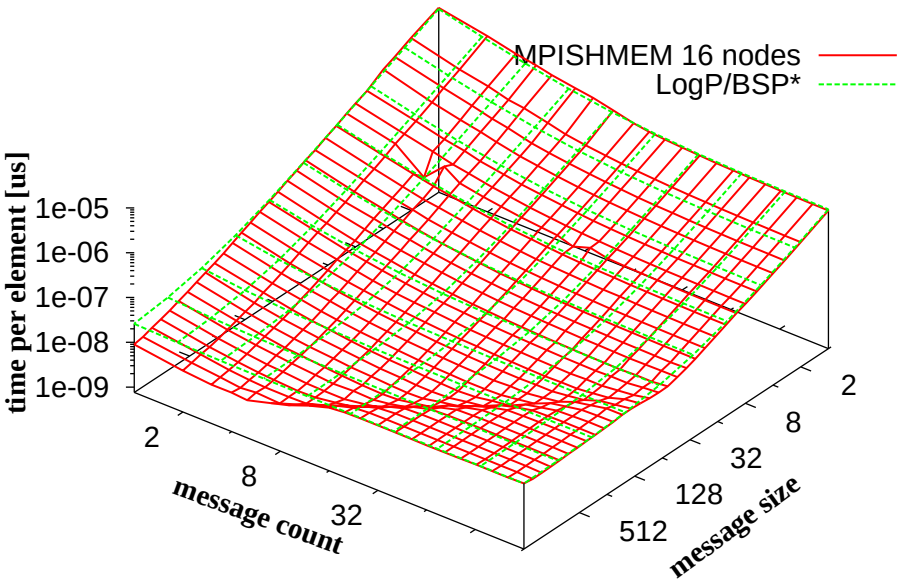
The overhead o is determined similarly to h_{half}, using weighted samples of g for the maximum message count and minimum message size g_{mm} (h^*_{min} is the smallest message size, for which samples were taken):

$$g_{mm} = \left(\sum \tilde{g}(h^*_{min}, c) \cdot c^2 \right) / \sum c^2 \ , \tag{3.5}$$

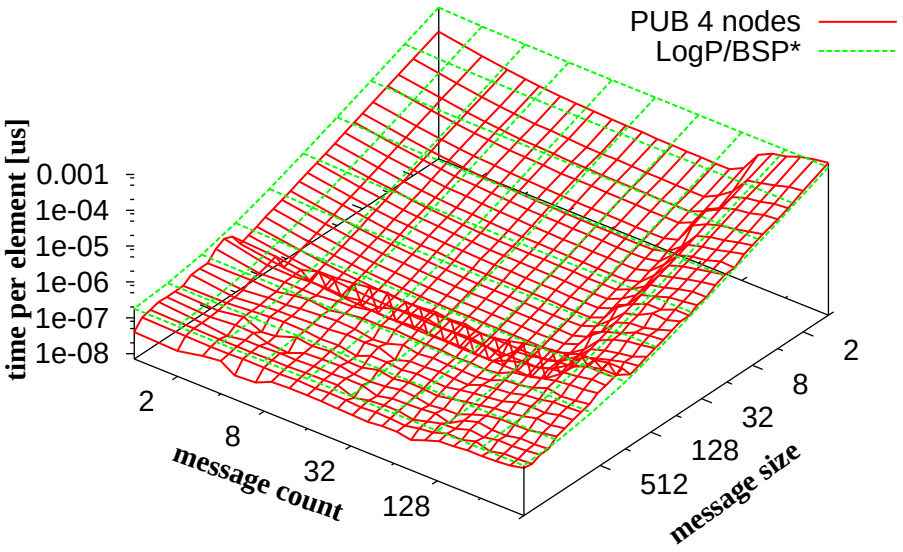
$$o = \left(\frac{g_{mm}}{g_\infty} - 1 \right) \cdot h^*_{min}. \tag{3.6}$$

Bandwidth Results on the Shared Memory System (`skua`)

On the shared memory system (`skua`), the best bandwidth (i.e. the lowest value of g) for `Put` and `Get` operations is achieved by the MPI-2 library; neither Oxtool nor PUB show comparable results. It can be seen in Figure 3.4(a) that the performance of these primitives for MPISHMEM has a local optimum at a certain communication cost, then the bandwidth drops and becomes constant. This behavior was only observed on the SGI machine and is caused by the limited size of its first level cache. The performance drop occurs approximately once the communication cost exceeds 16kBytes (this matches the level 1 data cache size on this system). Remote memory `Get` requests achieve a much better throughput for all-to-all communication, because they do not cause write conflicts and can make better use of the machine's cache. PUB shows a severe bandwidth drop when there is a large number of `(Hp)Get` requests. Also, the performance of these primitives is not very good compared to Oxtool's `(Hp)Get`. This can be seen in Figure 3.4(b), which shows that PUB suffers from

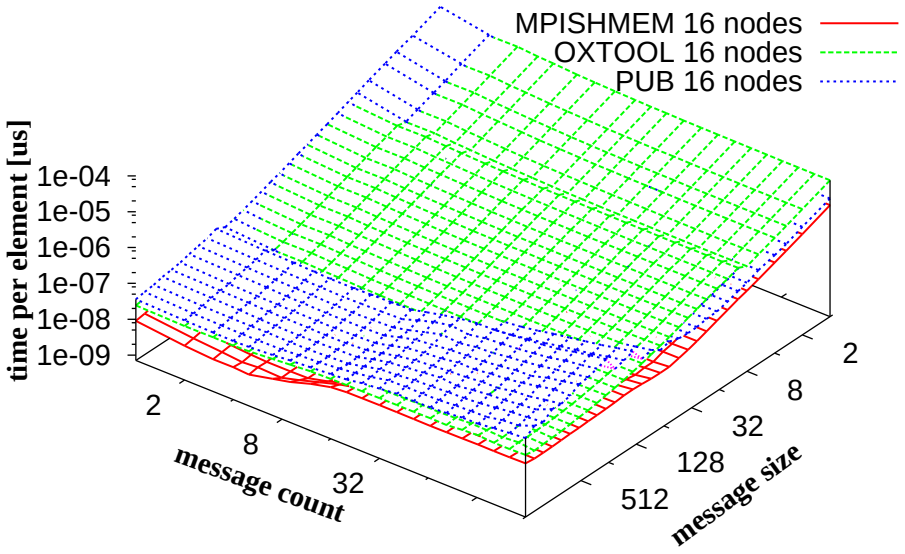


(a) `Put` (MPI-2)

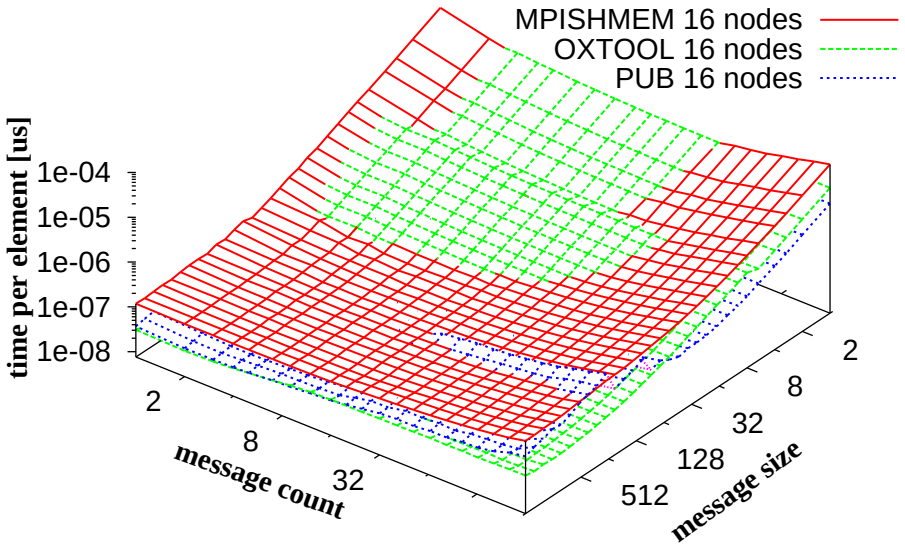


(b) `Get` (PUB)

Figure 3.4: Performance of all-to-all exchanges on `skua`



(a) `Put`



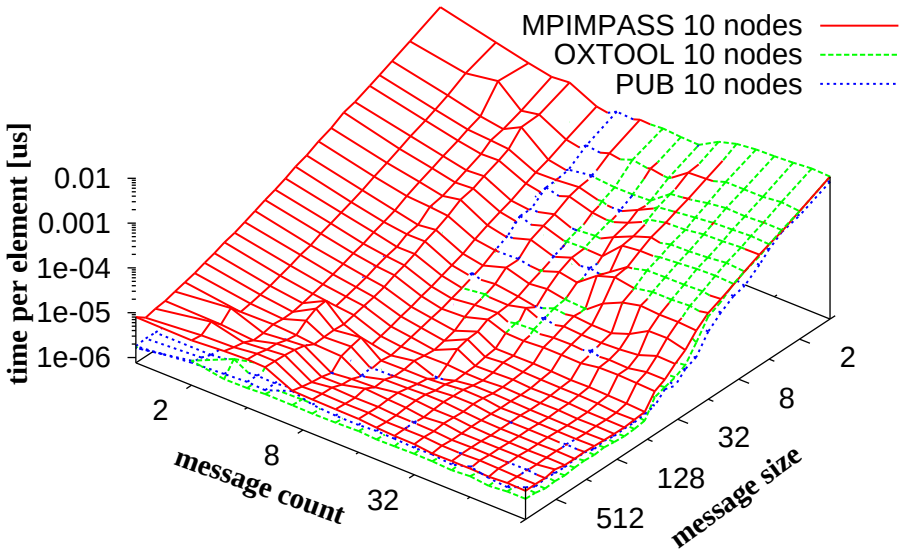
(b) `Send`

Figure 3.5: Performance comparison for all-to-all exchanges on `skua`

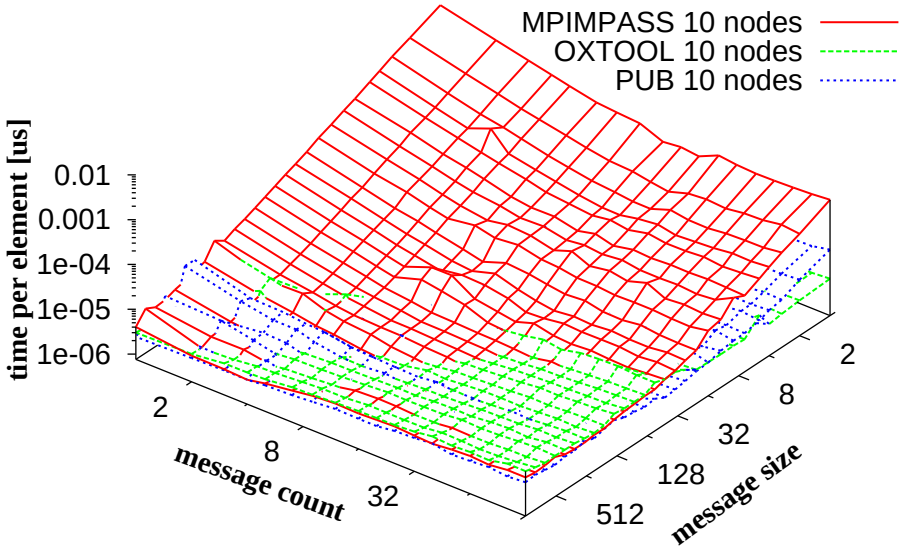
communication performance losses starting at a certain message size (approximately 256 doubles) and message count (approx. 128 messages). This behavior was only observed using `Get` and `HpGet` primitives. The performance differences between MPI-2's and Oxtool/PUB's `Put` primitives are less significant than between their `Get` functions. Using `Put`, MPI-2 shows the best performance for large communication costs. The performance of the `Send` primitive is best when using PUB or Oxtool, with slight advantages for PUB, although Oxtool shows better results for overhead (i.e. lower values of h_{half} and o). This shows that the message scheduling optimizations implemented by PUB and Oxtool can improve performance for large messages and all-to-all exchanges for many nodes using `Put` or `Send` (see Figure 3.5(b)). The overall behavior is similar on all numbers of processors for which samples were taken. A listing of all results on `skua` can be found in Appendix C, Tables C.1, C.2, C.3.

Bandwidth Results on Ethernet (`argus`)

On the Ethernet system (`argus`), PUB and Oxtool achieve similar values of g, both better than the ones obtained using MPI. For a larger number of processors and all-to-all communication, PUB has slight advantages. The per-message overhead o is particularly low for `(Hp)Put` operations (see Figure 3.6(b) and Table C.6). These primitives are obviously the best choice for implementations which should have predictable performance on this network. `Get` operations involve overhead for requesting data from the source node, thus the throughput does not increase much for a small request size and many requests (see Figure 3.6(a)). This cannot be avoided and should be considered when implementing algorithms which will have to run on a parallel computer using Ethernet. The best results on such a system can be achieved by using `Put` style communication. It can further be observed that the `Send` primitive of PUB has better performance and less per-message overhead than that of Oxtool (see Figure 3.7). Figure 3.8 shows that for random permutations using Oxtool, overhead exists for many small put requests (see the 'bump' on the right side of the graph). This occurred reproducibly when running on 10 nodes, and could not be observed for all-to-all exchanges. However, samples for all-to-all communication were only taken using up to 128 messages, as opposed to 512 messages for the random permutations.



(a) `Get` (all-to-all)



(b) `Put` (all-to-all)

Figure 3.6: Performance comparison on `argus` (`Put` and `Get`)

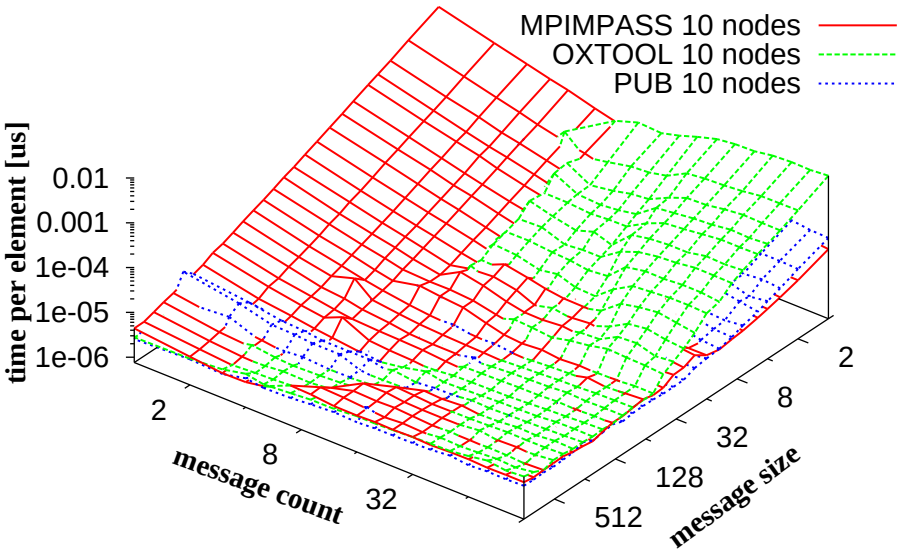


Figure 3.7: Performance comparison on `argus` (`Send`, all-to-all)

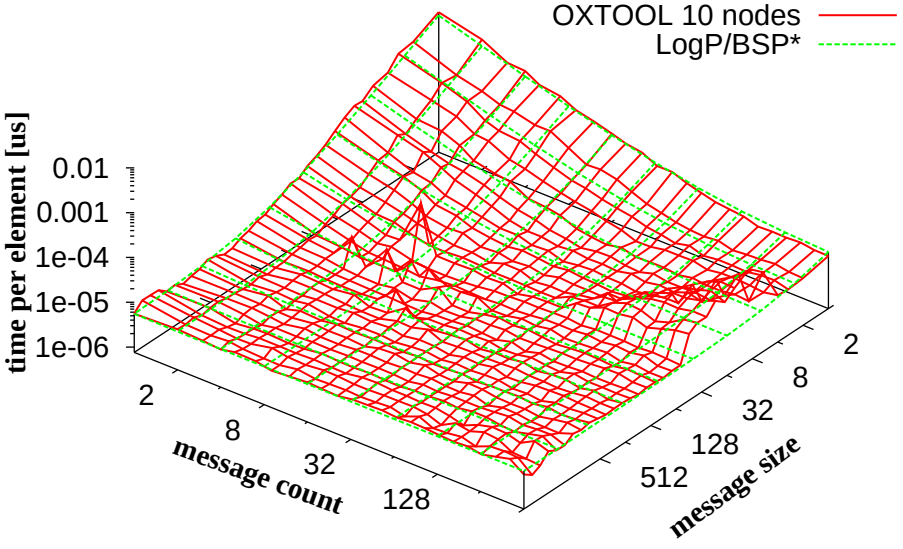
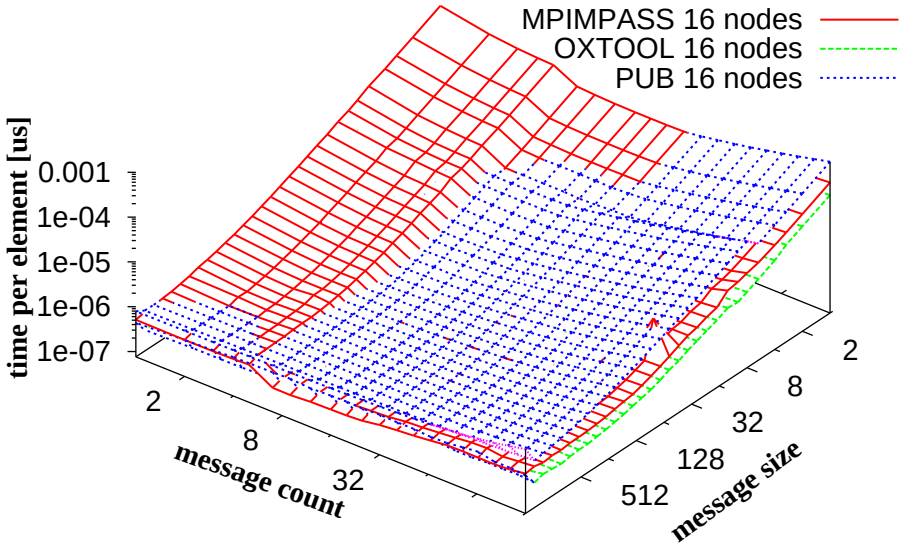
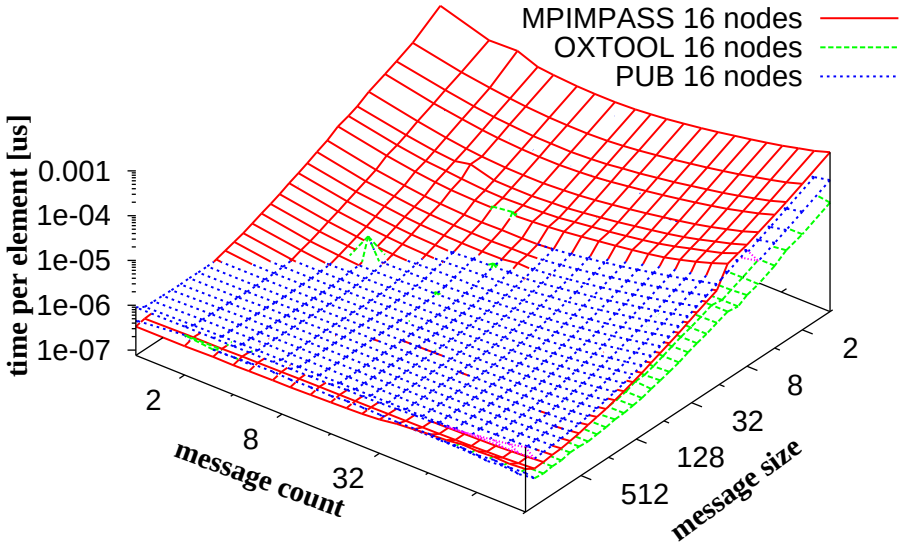


Figure 3.8: Performance on `argus`/Oxtool (`Put` (random permutation))



(a) `Get`



(b) `Put`

Figure 3.9: Performance comparison for all-to-all exchanges on `aracari` (`Put` and `Get`)

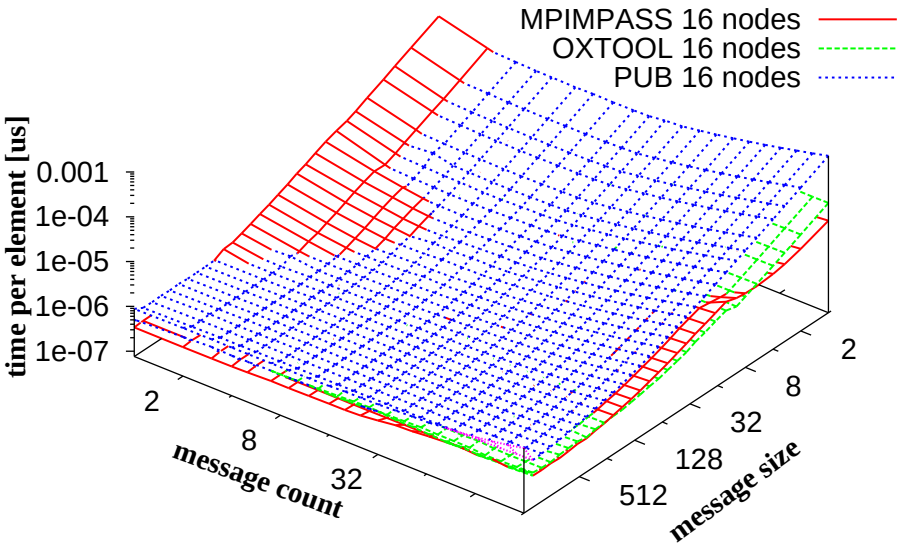


Figure 3.10: Performance comparison for all-to-all exchanges on `aracari` (`Send`)

Bandwidth Results on Myrinet (`aracari`)

The best results for the communication gap g on the Myrinet system (`aracari`) were achieved by Oxtool and MPI, with advantages for Oxtool. PUB has difficulties when the number of messages is large; the time per element rises slowly when the number of primitive calls increases. A similar behavior was observed in the experiments from [29]. Also, a clear 'step' can be seen in the surface of samples when PUB switches its message transfer method for messages too large for its message buffer (this problem and related issues with all the other communication libraries are discussed in Section 2.5). A slight drop in performance can also be observed on the Ethernet system, though it is less clearly visible there. For all primitives (see Figure 3.9 and Table C.9), the performance of Oxtool and PUB is better than for plain MPI when the communication volume (few and/or small messages) is low.

3.3 Measuring the Latency

The communication latency for the BSP computer can be determined by measuring the running time for the barrier synchronization function of the programming library. This includes

the time necessary for synchronizing all the processors, and for overhead necessary for receiving the data when buffered communication operations are used. Thus, it is not enough to measure the time for a plain call to the synchronization primitive of the communication library – it is also necessary to measure the time when this synchronization is preceded by an actual data transfer. The version of `bspprobe` from the Oxford BSP Toolset measures l for a plain synchronization without any data transfer (referred to as the 'low' synchronization latency) and for cyclic shift communication (referred to as the 'high' synchronization latency). Gerbessiotis et al. [29] argue that this is insufficient, since e.g. all-to-all communication can cause much higher synchronization latencies. Our benchmarking tool measures the latency for all-to-all communication as well. However, the latency could also vary when using different communication primitives for the data exchange that precedes the synchronization.

For performance prediction, values of l measured with communication are used whenever we assume that g is constant. If g is modeled using $g = g(h)$ or $g = g(h, h^*)$ (see Section 2.2), we assume the bandwidth parameters h_{half} and o describe this kind of overhead more exactly, and use the value measured without communication (which is equivalent to the parameter l_0 in Equation (2.4) on page 7). Therefore, we do not measure the synchronization latency for different communication primitives, as the parameters h_{half} and o are more significant for comparing overheads of different communication primitives.

Latency on Shared Memory (`skua`)

On the shared memory machine, Oxtool achieves the lowest latency among all libraries when communication precedes the synchronization. This is surprising, as PUB was compiled using an adapted version of its shared memory device and Oxtool – lacking adequate support for the SGI Altix shared memory architecture – had to be compiled on top of the message passing functions in MPI. The worst latency, except for all-to-all communication, is achieved by the 'naive' MPI-2 based BSP library. The latency results on `skua` can be found in Appendix C, Table C.2.

Latency on Ethernet (`argus`)

On the Ethernet system, the latency varies strongly between different runs, as the communication network can be more or less busy and different runs may be scheduled on different sets of nodes. Thus, only a general picture of latency can be obtained, which shows advantages for Oxtool and very long synchronization times for the MPI library. PUB shows an interesting behavior: the latency for all-to-all communication is lower than for a cyclic shift, as was observed in repeated measurements. As distributed memory systems connected by Ethernet are presumably the interconnection network for which Oxtool and PUB were most extensively tested and optimized, the gain of these libraries over the naive implementation is most visible. The latency results on `argus` can be found in Appendix C, Table C.5.

Latency on Myrinet (`aracari`)

In the benchmarks on the distributed memory/Myrinet system, PUB generally showed the lowest latency, Oxtool only showed lower latency when the number of processors was low. Compared to PUB, the MPI library needs 4–6 times as much time for a synchronization. The complete results of the latency measurements on `aracari` can be found in the Appendix (Table C.8).

Chapter 4

Matrix Multiplication

Dense matrix multiplication has been chosen as our first practical benchmark. This is for several reasons: first, the problem and algorithms are relatively simple and well understood. The number of operations necessary, as well as the theoretical memory requirements, are easy to calculate. Furthermore, matrix multiplication is of relevance for many applications in scientific computation. Consequently, this problem can serve as a good practical benchmark for comparing the predictability and absolute performance that can be achieved by BSP algorithms using different communication libraries.

The memory efficient matrix multiplication algorithm described in [55] was implemented in order to study experimentally the effects that different block sizes and data distributions have on its running time. Further, its performance was compared to PBLAS [74], which includes a standard parallel matrix multiplication implementation. Benchmarks using a matrix multiplication algorithm were also conducted in [29]; however, their algorithm differs from ours as it generates a different communication pattern and is not memory-efficient. Also, their implementation uses an IJK loop for the local products, which leads to longer local computation times. We use BLAS, which has better performance and is less dependent on the CPU cache size.

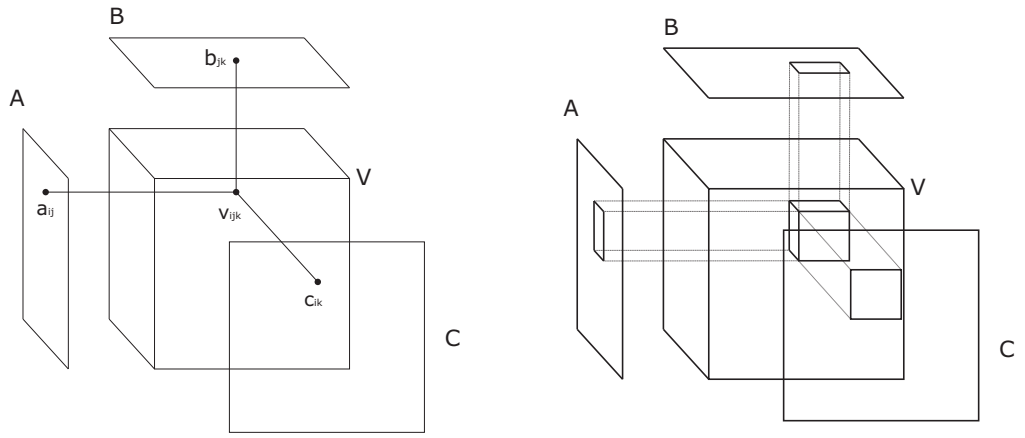


Figure 4.1: Matrix-multiplication graph and block partitioning

4.1 The Algorithm

A simple sequential algorithm for computing the matrix product

$$C = A \cdot B \tag{4.1}$$

of two dense $n \times n$ matrices A and B uses the formula

$$c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}, \text{ having } A = [a_{ij}], \quad B = [b_{ij}], \quad C = [c_{ij}] \ , \tag{4.2}$$

with $i, j = 1, 2, ..., n$. This is equivalent to representing the elementary products as points of an $n \times n \times n$ cube, with data broadcasting in two dimensions and data combining by addition in the third dimension. The basic idea for the parallel algorithm is to partition the problem into cubic blocks, which correspond to products of smaller matrices. These products can be computed locally on each processor (see Fig. 4.1). The smaller these blocks are, the less temporary storage per processor will be used. A smaller block size increases the required number of supersteps and decreases the communication cost per superstep. The communication pattern of the algorithm can be controlled by modifying the block size parameter. For this reason, it seems to be a good choice for studying the differences in performance of BSP-style communication library implementations.

The matrix multiplication cube is partitioned into a number of blocks q. For simplicity,

we assume that $q^{1/3}$ and $n/q^{1/3}$ are integers. Furthermore, we use values of $q > p$ for maintaining memory efficiency. These blocks are processed locally on each processor, using the sequential matrix multiplication functionality from level 3 BLAS. Each cubic block defines an intermediate product V_{IJK}, which is defined as

$$V_{IJK} = A_{IJ} \cdot B_{JK} \text{ with } 1 \leq I, J, K \leq q^{1/3} \ . \tag{4.3}$$

A_{IJ} and B_{JK} are the square blocks in A and B at position $s \cdot (I-1)+1, s \cdot (J-1)+1$ and $s \cdot (J-1)+1, s \cdot (K-1)+1$ of width $s = n/q^{1/3}$. The temporary results V_{IJK} are added to obtain the result matrix C:

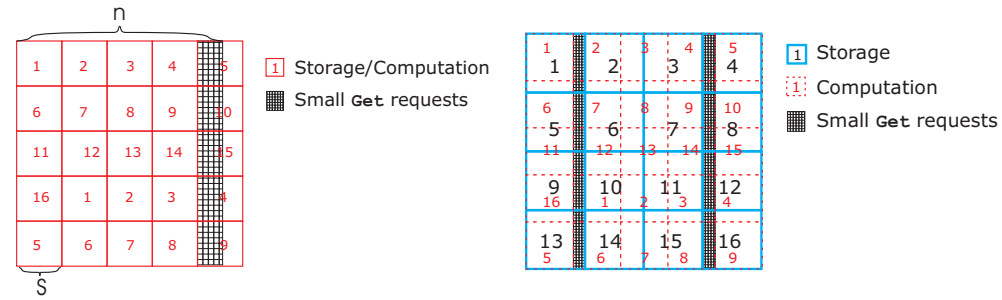
$$C_{IK} = \sum_{J=1}^{q^{1/3}} V_{IJK} \qquad \text{with } 1 \leq I, K \leq q^{1/3} \ . \tag{4.4}$$

Hence, the parallel algorithm has three basic steps for computing each intermediate product V_{IJK}: an input phase for getting the parts of both matrices, followed by local computations, and finally, an output phase to redistribute the result data.

4.2 Input/Output Data Distributions

Customized Data Distribution

For the first set of experiments, the initial data is predistributed block-cyclically by square blocks of size $n/q^{1/3}$. Theoretically, this should lead to the best performance, as the overhead for getting the input data in each superstep is minimal. Also, when assigning the computation of C_{IK} block-cyclically, no output phase is necessary when $q^{2/3} \bmod p = 0$, because the processor which computes all the values of V_{IJK} in one block column $J = 1, ..., q^{1/3}$ also stores the corresponding block C_{IK}. If $q^{2/3} \bmod p > 0$, some block columns remain that are not computed by the processor storing the result, and an output phase is necessary in approximately $\lceil (q^{2/3} \bmod p) \cdot q^{1/3}/p \rceil$ supersteps. In the input phase, each processor has to get two input blocks of size $s \times s$. The input data has to be distributed in every superstep to retain memory efficiency. For each logical superstep, the implementation uses



Small messages when n mod $s > 0$ Small messages when s mod $\sqrt{p} > 0$

Figure 4.2: Overpartitioning and matrix data distributions on 16 processors

two synchronizations; one for the input and one for the output phase. This yields the following computation, communication and synchronization costs:

$$W = \left\lceil \frac{q}{p} \right\rceil \cdot \frac{n^3}{q} , \tag{4.5}$$

$$H = \left\lceil \frac{q}{p} \right\rceil \cdot \frac{n^2}{q^{2/3}} \cdot (2 + \lceil (q^{2/3} \bmod p) \cdot q^{1/3}/p \rceil) , \tag{4.6}$$

$$S = \left\lceil \frac{q}{p} \right\rceil \cdot 2 . \tag{4.7}$$

The overall running time can thus be computed like this:

$$\begin{aligned} T = \; & f \left\lceil \frac{q}{p} \right\rceil \cdot s^3 + \\ & g \left\lceil \frac{q}{p} \right\rceil \cdot s^2 \cdot (2 + \lceil (q^{2/3} \bmod p) \cdot q^{1/3}/p \rceil) + \\ & l \left\lceil \frac{q}{p} \right\rceil \cdot 2 . \end{aligned} \tag{4.8}$$

When using h_{half}, a communication size $h^{in} = 2s^2$ for the input phases and $h^{out} = s^2$ for the output phases can be assumed. When using the variable bandwidth approximation $g = g(h, h^*)$, the average message size h^* can be assumed to be s^2. However, if n mod $s > 0$, there are $q^{2/3}$ supersteps in which $h^{in} = s(n \bmod s)$, since the last block column has smaller blocks, which are transferred row by row. Per-message overhead then causes longer running times when n is not a perfect multiple of s.

Static Data Distribution

For the second set of experiments, input and output data are distributed statically in a 2-dimensional grid of blocks sized $n/\sqrt{p} \times n/\sqrt{p}$. The modified BSP running time for this data distribution is:

$$\begin{aligned} T' = \quad & f \quad \left\lceil \frac{q}{p} \right\rceil \cdot s^3 + \\ & g \quad \left\lceil \frac{q}{p} \right\rceil \cdot s^2 \cdot \left(2 + 1/q^{1/3} + \lceil (q^{2/3} \bmod p) \cdot q^{1/3}/p \rceil \right) + \\ & l \quad \left\lceil \frac{q}{p} \right\rceil \cdot 2 \ . \end{aligned} \tag{4.9}$$

We now have to account for the fact that an output phase is necessary every $q^{1/3}$ supersteps. The intermediate results can be added up and buffered locally, but have to be sent to their final position every time a block column V_{IJK} is completed. For transmitting the parts of A and B, the DRMA `HpGet` primitive of the libraries is used. When $\sqrt{p} \neq q^{1/3}$, the matrices are transferred row by row. This leads to smaller messages/`HpGet` requests as the number of blocks q increases, and can create a more irregular communication pattern. To use $g = g(h, h^*)$ and to take per-message overhead into account, the communication time can be estimated as:

$$\begin{aligned} h_i &= 2s^2 \qquad , h_o = s^2 \\ T_{comm} &= g(h_o, h^*) \cdot h_o \cdot \left(1/q^{1/3} + \lceil (q^{2/3} \bmod p) \cdot q^{1/3}/p \rceil \right) + \\ & \quad g(h_i, h^*) \cdot h_i \ . \end{aligned} \tag{4.10}$$

The message size h^* is assumed to be the average of all possible message sizes for a given combination of s, n and $\sqrt{p}$. When $n/\sqrt{p}$ is a perfect multiple of s, complete rows of size s can be transferred, otherwise much smaller messages arise when a row is fetched from two different processors. This causes unsteady performance when q is large. The same effect can lead to small messages when the matrix size n is not a perfect multiple of s. In this case, a whole block column has rows with width n mod s for both the input matrices and the output matrix. This causes unsteady performance when q is small. For larger values of q,

the value of s decreases, and so does the remainder n mod s. However, these effects should be suppressed or at least reduced, if the BSP library implements efficient message combining functionality. Figure 4.2 shows an example where the small messages arise for both data distributions when using $q^{2/3} = 25$ and $p = 16$.

4.3 Experiments

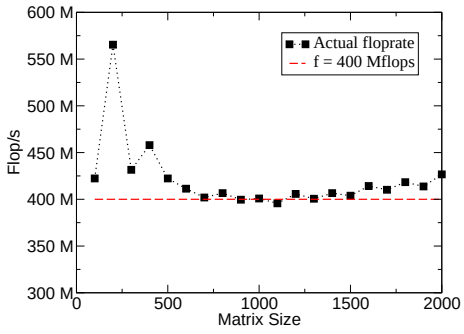


Figure 4.3: Matrix multiplication — individual processor flop rate F (`aracari`)

System	f	$F = 1/f$
`skua`	0.33 ns/flop	3 G flop/s
`argus`	0.83 ns/flop	1.2 G flop/s
`aracari`	2.5 ns/flop	400 M flop/s

Table 4.2: Matrix multiplication — Experimental values of f

The experiments were conducted on the same systems as introduced in Section 2.4. For each system and library, we ran the matrix-multiplication algorithm for matrix sizes from 100×100 to 2000×2000 and measured the running time. The number of blocks in the cube was set to the values $q = 3^3, 4^3, 5^3, ..., 9^3$. For larger numbers of processors, the values $q = 3^3, 4^3$ were left out if they would lead to underpartitioning (making sure that $q^{2/3} > p$). In order to compare the performance to PBLAS, samples were also taken for matrices of sizes up to 10000×10000 using the smallest possible value of q relative to each number of processors. The values of f were obtained by measuring the computation time separately in one run (see Figure 4.3). Table 4.2 contains the values that were used for the predictions.

To evaluate the performance of the libraries, we compare the speedup

$$S = T(p)/T(1) \tag{4.11}$$

obtained for different problem sizes, where $T(p)$ denotes the running time of the algorithm on p processors. All speedups shown here are relative to the running time of the same algorithm with the same number of overpartitioning blocks on only one processor, in order to show how much communication overhead is caused by the communication library. However, if a realistic scalability analysis[1] is to be performed, the performance should be compared to the fastest sequential implementation available on each machine. Therefore, we also list the peak efficiency (the effective flop rate divided by one processor's asymptotic flop rate) achieved by each library in Appendices E and D:

$$E = \frac{n^3}{T(p) \cdot p} / F \ . \tag{4.12}$$

The running time was averaged over 5 runs for each value of q and n. MPI occasionally showed spikes of very long communication time, which occurred randomly and were probably due to different traffic on the communications network. For all predictions, the values corresponding to all-to-all exchanges using the `HpGet` primitive were used for the parameters g, h_{half}, o and l. This matches the communication pattern of the implementation, assuming that every processor stores an equal fraction of the input data. The contribution of the latency l to the running time of the algorithm can be considered negligible in case of the shared memory machine (`skua`) and the Myrinet cluster (`aracari`). Even at the maximum number of blocks (729) and minimum number of processors (4), there are only 183 supersteps. Assuming a worst case latency of 2000 μs, the synchronization time only adds up to 0.37 s. In general, l only becomes significant when the communication network has high latency and the number of supersteps is large[2]. Using only $g = g(h)$ for prediction does not improve prediction accuracy, as the communication size per superstep h is always large and increases quadratically with the matrix size. When including a per-message overhead in the performance model, the predictions show spikes at the same positions as the measurements. Nevertheless, even when using $g = g(h, h^*)$ to predict the communication times, the theoretical values deviate from the measurements in many cases. Reasons for that are:

[1] Other dense matrix multiplication benchmarks on different parallel systems can be found in [64, 43].

[2] This can be seen well from the results on the Ethernet cluster (`argus`), where no speedup is achieved already when $q > 8$ (see e.g. Figure 4.6(a)). The latency on `argus` is approximately twenty times as large as on `aracari`.

- The theoretical approximation of g deviates from the real bandwidth values. Particularly on some of our parallel machines, the bandwidth also depends on factors which were not taken into account, such as the first level cache size on `skua`. Also, overhead increases with message size and count on certain system/library combinations.

- The benchmarking experiments on which the predictions are based measured the transfer time for messages of equal size. This is not necessarily true for the matrix multiplication algorithm. Depending on the data distribution, messages of different size can arise in the same superstep.

- For small overall communication and message sizes, the bandwidth approximation presumably has the largest error. This error propagates quadratically to the communication time estimation, as communication costs increase quadratically with the problem size.

4.4 Results on Customized Data Distribution

The following results were obtained by running the matrix multiplication algorithm on matrices predistributed in the same blocks as used for computation. As discussed in the previous Section, this constitutes a simple communication pattern and large messages.

Results on Shared Memory (`skua`)

On `skua` (Figures 4.4 and 4.5), MPI-2 shows the best overall performance, and Oxtool has slight advantages for very small matrices. PUB achieves the lowest performance, especially on 4 nodes. These results can by explained by looking at the communication performance for all-to-all exchanges using `HpGet`. PUB's `HpGet` primitive induces increasing overhead for large messages. This particularly affects performance on a lower number of processors, i.e. when the largest data blocks have to be transferred. If the number of processors increases, PUB and Oxtool show greater sensitivity to the case when n mod $s > 0$, which causes $q^{2/3}$ blocks that have to be transferred line by line. The overhead induced by this effect causes the spikes that can be seen in Figure 4.4. The best absolute scalability and highest efficiency

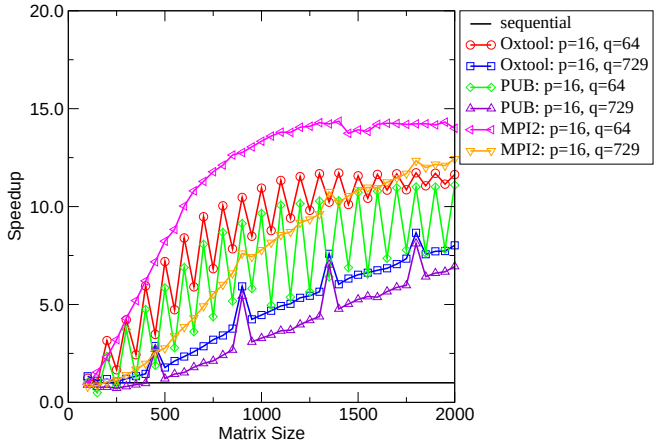


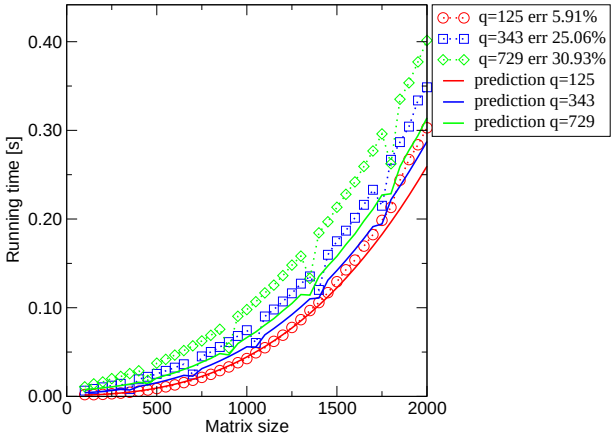
Figure 4.4: Customized data distribution — Speedup on `skua` (using 16 processors)

is achieved by MPI-2.

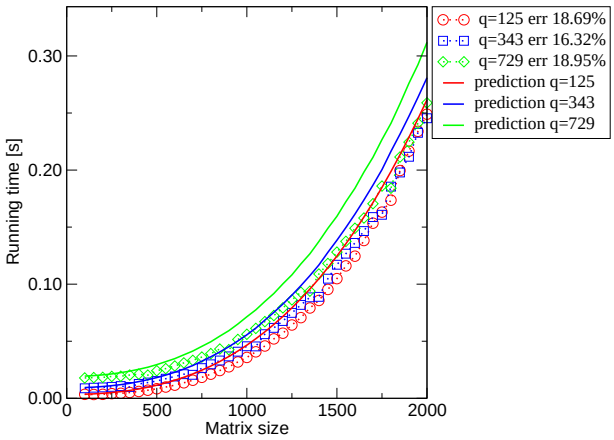
The runtime predictability on `skua` is best when using Oxtool or MPI. The predicted runtime for MPI is higher than measured, because MPI has a local performance optimum for a communication size of 16kB per superstep, which is not covered by the performance model. Oxtool has increasing overhead for higher numbers of messages. This is not included in our model, therefore the runtime predictions for Oxtool and large values of q and n are smaller than the measured times. PUB shows the least predictable results, and also the greatest sensitivity to the communication imbalances introduced by the data distribution on this system. Figure 4.5 shows an overview of predictions and measured running times on 16 processors. Performance predictability decreases when using a higher number of processors, but is still acceptable. When using $g = g(h, h^*)$, the mean relative error for MPI-2 is below 20% in most cases.

Results on Distributed Memory, Ethernet (`argus`)

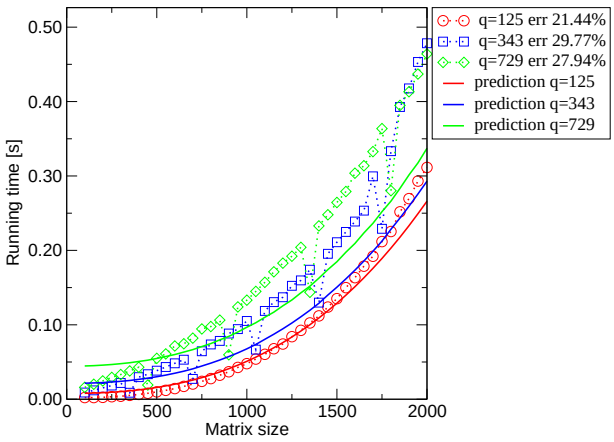
On `argus`, both of the optimized libraries clearly show better performance than MPI, in particular when using many CPUs and when transferring small messages. Because `mpich-p4 1.2.5` is the only MPI implementation available on this machine, measurements for $p = 4$ and $q = 8$ could not be run with the MPI library, since they caused too large MPI messages (see Section 2.5 for further discussion). Especially when the communication cost is large, or



(a) $g = g(h, h^*)$, Oxtool

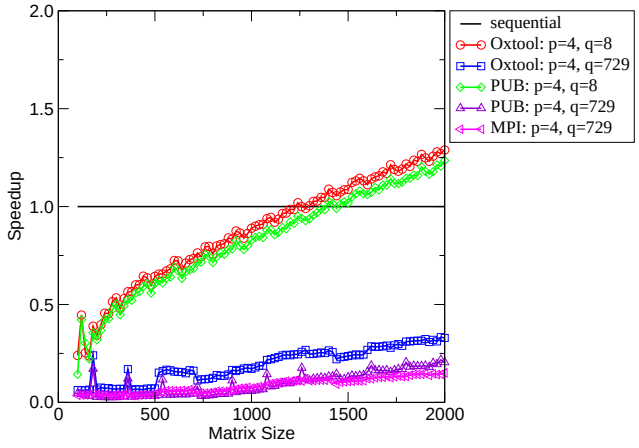


(b) Fixed value of g, MPI

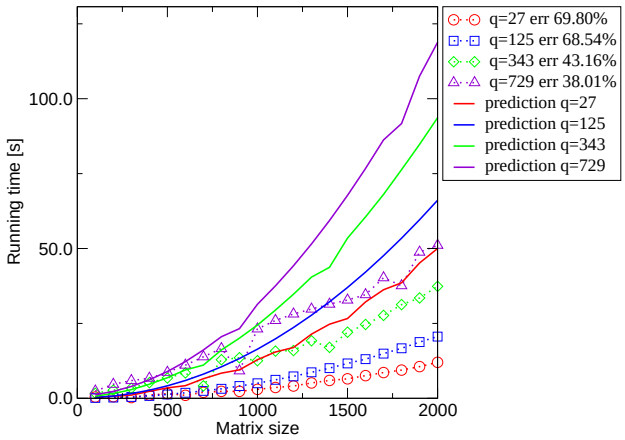


(c) Fixed value of g, PUB

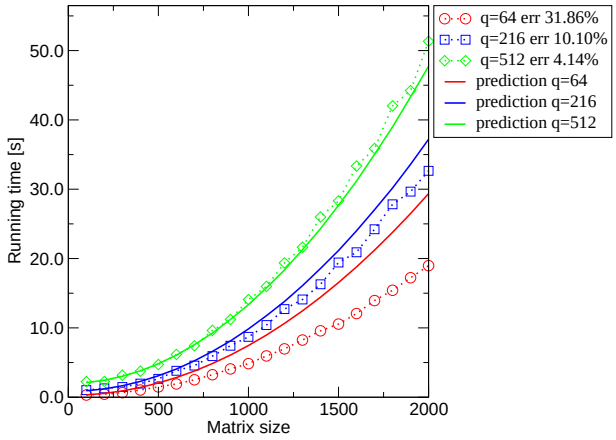
Figure 4.5: Customized data distribution — Predictability on `skua`, 16 processors



(a) Speedup on 4 processors



(b) $g = g(h, h^*)$, PUB, p=4



(c) $g = g(h, h^*)$, MPI, p=10

Figure 4.6: Customized data distribution — Results on `argus`

when there are many synchronizations (i.e. for large values of q), the matrix multiplication algorithm implementation benefits from using an optimized library. There are advantages for Oxtool if the message size is small. Predictably, there is no or very little speedup for the matrix sizes under consideration, and the efficiency is below 10% in all experiments.

The predicted running times on 4 processors are much larger than the measured ones. Only the predictions for MPI match reasonably well. This is presumably caused by the fact that the effective value of g_∞ is lower than the one measured by our benchmarking routine. As `argus` is an SMP system, communication between two processors in the same SMP node takes much less time than communication between different nodes using Ethernet. When the number of processors is low, many of the block transfers can thus be carried out much faster than predicted by the benchmark results.

Results on Distributed Memory, Myrinet (`aracari`)

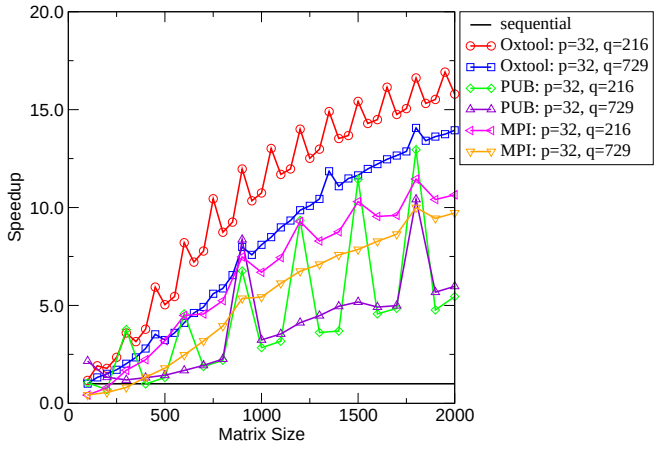
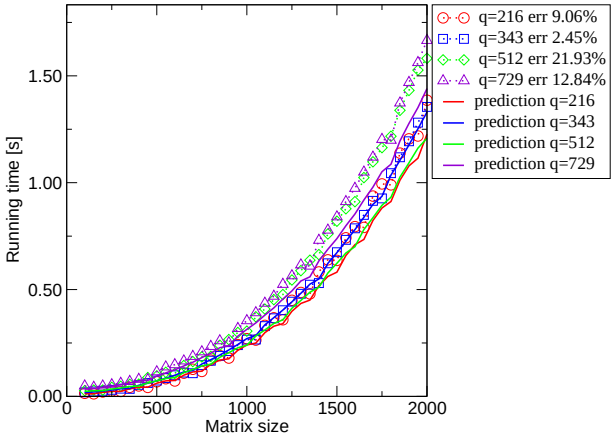


Figure 4.7: Customized data distribution — Speedup on `aracari` (using 32 processors)

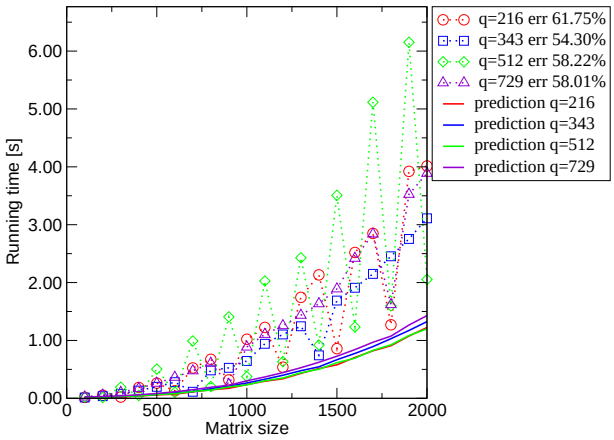
Better speedup than on `argus` was achieved on `aracari` (see Figure 4.7). Oxtool achieves the best scalability for all values of q. For small values of q, PUB and MPI show similar results. When q becomes larger, there are advantages for PUB. When p increases, the performance of PUB and MPI's drops drastically for values of n that are not a perfect multiple of s. MPI clearly has better performance when the message size is large, PUB performs better when the message size is small. The best run time predictability is achieved by Oxtool (see Figure 4.8), especially when using $g = g(h, h^*)$, in which case the predictions are good for all considered numbers of processors. MPI only produces predictable results if few nodes are used and per-message overhead is taken into account ($g = g(h, h^*)$). PUB has the least predictable performance; obviously the bandwidth gap for PUB shows characteristics which are not modeled by our approximation of g (see Chapter 3 for a discussion).

4.5 Results on Static Data Distribution

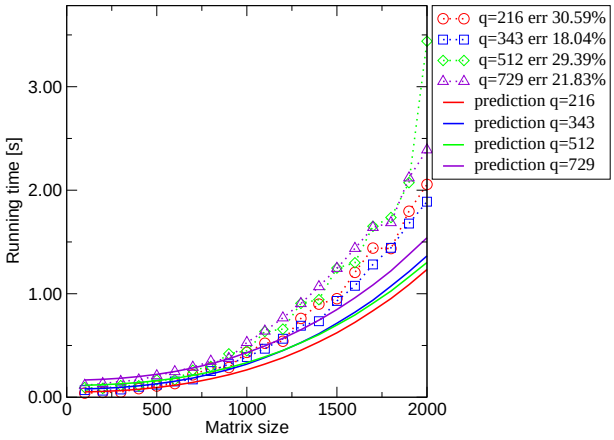
The following results were obtained by running the matrix-multiplication algorithm on matrices distributed statically with storage block size n^2/p. The matrices are transferred row by row, hence the message size is much smaller than it is using predistributed data blocks. This can be seen as a more difficult benchmark for BSP libraries, as only those with stable



(a) $g = g(h, h^*)$, Oxtool

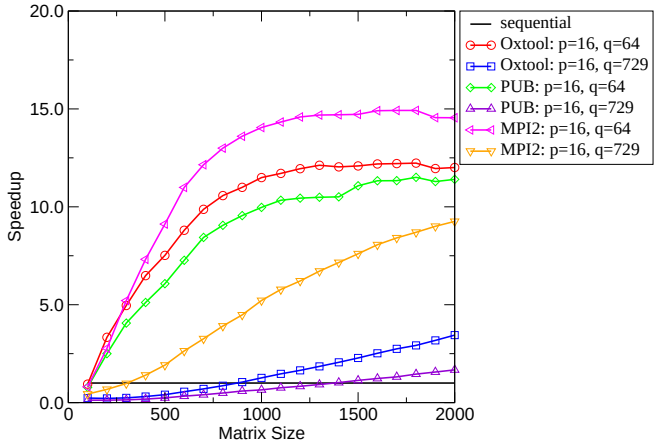


(b) $g = g(h, h^*)$, PUB

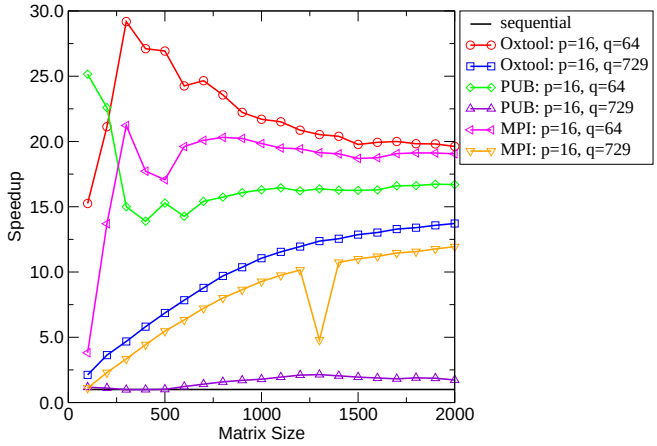


(c) Fixed value of g, MPI

Figure 4.8: Customized data distribution — Predictability on `aracari` (32 processors)

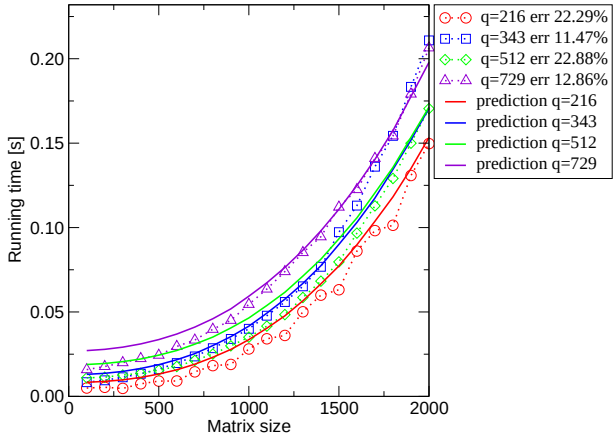


(a) `skua` using 16 processors

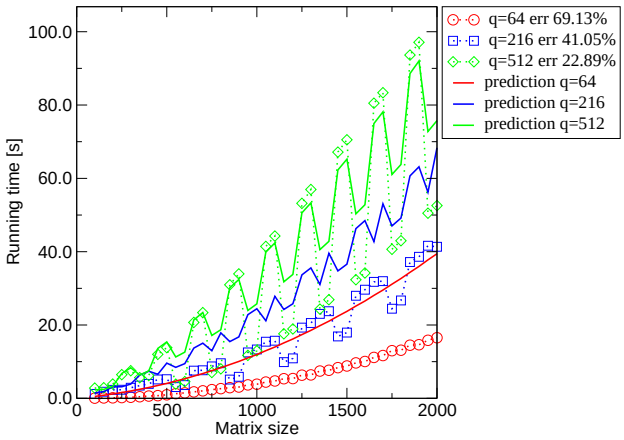


(b) `aracari` using 16 processors

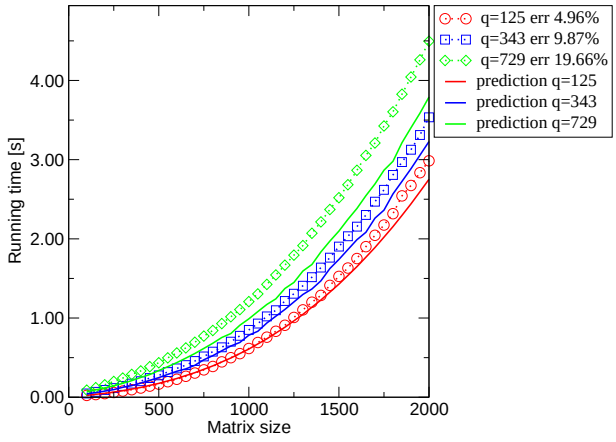
Figure 4.9: Static data distribution — Speedup



(a) `skua`/MPI, fixed value of g (32 processors)



(b) `argus`/Oxtool, $g = g(h, h^*)$ (10 processors)



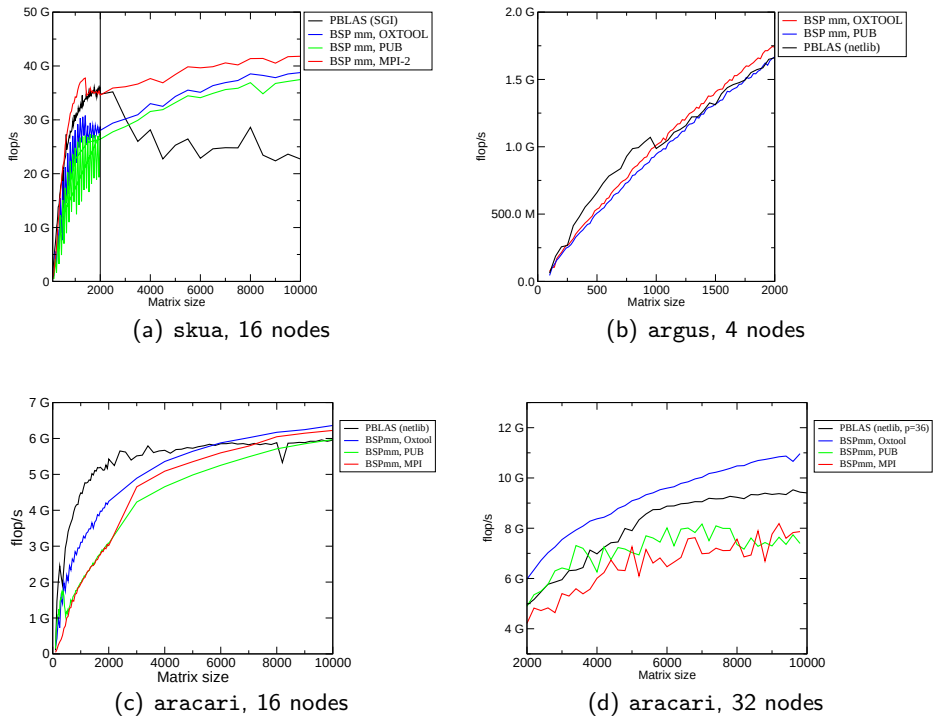
(c) `aracari`/Oxtool, $g = g(h, h^*)$ (16 processors)

Figure 4.10: Static data distribution — Predictability

performance for small messages are able to achieve good performance. Thus, the major difference between the results for the customized data distribution, and those for static data distribution, are slightly longer running times on all systems. The efficiency is lower, particularly for small subproblem sizes communication time takes up a higher percentage of the overall running time. Because of this, there is no speedup at all on `argus`, although the overall behavior on `argus` remains the same: PUB and Oxtool perform similarly, and better than MPI. On `skua` and `aracari`, PUB showed worse performance with this data distribution, because it has high per-message overhead for `HpGet`. Also on `aracari`, performance and efficiency of MPI improved in some cases and became more stable compared to the experiments with the customized data distribution. Runtime predictability using a constant value of g is good only if the number of processors and q are low. Otherwise, per-message overhead is introduced by the fact that there are more small `HpGet` requests in every superstep, than there are when the data is predistributed. On all systems, predictability improves when using $g = g(h, h^*)$, but is still not as good as for the customized data distribution. Especially PUB, having increasing values of g as a result of higher per-message overhead produces running times far beyond the predicted values. However, as in Section 4.4, the runtime behavior and spikes caused by per-message overhead can be predicted qualitatively on most systems.

4.6 Comparison with PBLAS

For evaluating the absolute performance of our BSP algorithm, the flop rate achieved using the customized data distribution was compared to the performance of PBLAS. The PBLAS [74] library includes an optimized parallel matrix multiplication implementation and is available on most parallel computers. Figure 4.11 shows the results for all systems (notice that between $n = 2000$ and $n = 10000$, the running time is only sampled in steps of $\Delta n = 500$. Before $n = 2000$, the step size is smaller, hence Figure 4.11(a) shows a marker at $n = 2000$). The effective flop rate $F = n^3/T$ (T denotes the running time) achieved by the BSP algorithm using the smallest value of q is compared to the effective flop rate achieved by PBLAS for the same matrix sizes. As can be seen, our BSP algorithm only achieves comparable



(a) `skua`, 16 nodes

(b) `argus`, 4 nodes

(c) `aracari`, 16 nodes

(d) `aracari`, 32 nodes

Figure 4.11: Performance comparison with PBLAS

performance to PBLAS when using the best suited BSP library for each system.

On the shared memory machine, only the MPI-2 version achieves better or equal performance, compared to PBLAS. The flop rate for PUB and Oxtool shows oscillations caused by the data alignment effects that were described in Section 4.1. For large matrices, the performance achieved by PBLAS decreases, whereas for our implementations it remains stable.

When using 4 processors on the Ethernet cluster, both PUB and Oxtool achieve a slightly better flop rate than PBLAS above a certain matrix size. Below that size, the performance is lower but comparable. For larger matrices and higher numbers of processors, PBLAS achieves much better performance on this system. PBLAS is optimized for reducing the communication cost involved in the matrix multiplication [14], whereas our algorithm is optimized for memory efficiency and involves all-to-all style communication. This is obviously not well suited for being used on Ethernet. Thus, the BSP algorithm only performs well when using few processors on this system. Since `argus` is an SMP system, communication is still relatively inexpensive when using 4 processors, as communication between processor pairs in

one SMP node is much cheaper than communication using Ethernet.

On `aracari`, the BSP implementations achieve better performance than PBLAS with large matrices or when the number of processors is high. Only with small matrices or when the number of processors is lower does PBLAS achieve better performance than our implementation.

4.7 Experiment Summary

The performance of our memory-efficient dense matrix-matrix multiplication algorithm was studied for two different input and output data distributions. As expected, the performance was better when customizing the data distribution to reduce per-message overhead. However, when using the best performing BSP library on the systems with the fast communication networks (Myrinet or shared memory), the running time remained predictable and scalability was also achieved when using the static data distribution, which induced a more complicated communication pattern. On the Ethernet system, performance was only good using few processors. When using higher numbers of processors, the communication cost became too high to achieve scalability, as larger parts of the data were transferred using Ethernet rather than between CPU pairs on a SMP node. This shows that BSP matrix multiplication implementations can achieve good scalability and do not strongly depend on the data distribution or communication network when the communication library is suitable and the communication network has good performance. On message passing systems, an implementation using `Put`-like primitives will presumably achieve better performance, as these have less per-message overhead. On the other hand, the `Get`-like communication we used is better suited for shared memory systems, where it induces a memory access pattern that makes best use of the CPU cache hierarchy. Furthermore, the use of `HpGet` suggests itself for implementing memory efficient algorithms with variable input distribution: the `HpGet` primitive does not involve buffering and `Get`-like communication is an intuitive way for implementing different data distributions.

The best results can be seen on the shared memory machine (`skua`), where MPI achieves the best scalability, because its `HpGet` communication primitive is directly translated to

remote memory read operations by SGI's MPI-2 implementation. Oxtool runs on top of the message passing functionality in MPI, which causes more overhead for transferring data. PUB shows the worst performance on this system: it transfers the data using a packet mechanism that needs at least two memory copy operations per data element, and induces further overhead when the communication size exceeds a certain limit.

All implementations except for PUB achieved good speedup on `aracari` (Myrinet). PUB was only able to deliver good speedup when the communication volume was small (in particular the number of messages), otherwise the performance decreased. The best performance was achieved by Oxtool and the MPI implementation. For higher values of q and small matrix sizes, superlinear speedup can occur when the local product blocks fit into the CPU cache, thus reducing local computation times. Notice that, this speedup is only superlinear in relation to the sequential version that uses the same data layout, and thus the same overhead for the input phase. Nevertheless, the efficiency results in Appendices E and D show that very good efficiency of up to 90% is achieved on `aracari` when using 4 processors.

On `argus` (Ethernet), none of the libraries achieved good speedup. For the given matrix sizes, individual nodes usually were able to perform the multiplication faster than many nodes in parallel. This is due to the fact that the individual nodes are very fast and the communication network is slow. The communication time thus becomes the predominant part of the overall running time. Another factor is the overhead caused by running on top of MPI. However, PUB and Oxtool showed an advantage over plain MPI, having better and more stable communication performance throughout all experiments.

On the Ethernet system the approach for minimizing communication cost employed in PBLAS is more effective. Even the use of an optimized library like Oxtool or PUB for our simple BSP algorithm does not yield comparable performance. However, the best BSP libraries on the systems using fast communication networks (MPI-2 on `skua` and Oxtool on `aracari`) always achieved better results for large matrices. Implementing a more efficient BSP algorithm for matrix multiplication, for example, a parallel variant of Strassen's algorithm (see e.g. [64, 52, 55]), could further improve the performance results.

Chapter 5

Longest Common Subsequence Computation

As a second realistic benchmark for BSP libraries, we chose the problem of computing the length of the longest common subsequence (LLCS) of two strings. Computing the LLCS, like matrix multiplication, is a well-studied problem. It can be solved using a simple dynamic programming algorithm [15], and parallelized on a BSP computer using a wavefront approach [30, 4]. For our purposes, we extend this approach using the same method as in [3] to use a variable block size for parallel dynamic programming, which improves the performance especially when the problem size is large. This method is similar to the overpartitioning approach for achieving memory efficiency that was used in the previous Chapter. We study this BSP dynamic programming approach and give a performance model for predicting the running time. For sequential computation, both a linear space dynamic programming approach and a more efficient bit-parallel algorithm were implemented. A survey of bit-parallel algorithms for various kinds of string comparison can be found in [58]. Further related publications are [18, 45, 46]. Sequential approaches to extract the LCS are proposed in [41, 44]. Crochemore et al. recently adapted these sequential algorithms to use bit-parallel computation [17].

In our experiments, we only compute the length of the LCS. The LCS itself can be obtained in a post-processing step. For a simple LLCS algorithm (without bit-parallel computation), such a post-processing step is described in [26]. Extracting the actual longest common

subsequence using bit-parallelism can be done in the same asymptotic time by saving the whole dynamic programming matrix and then running a second sweep of this matrix, using the method from [16]. A parallel (but non-BSP) LCS algorithm using bit-parallelism and a linear processor array is studied in [56].

5.1 Problem Definition and Simple Algorithm

Let $X = x_1x_2 \ldots x_m$ and $Y = y_1y_2 \ldots y_n$ be two strings on an alphabet Σ of constant size σ. A subsequence U of a string is defined as any string which can be obtained by deleting zero or more elements from it, i.e. U is a subsequence of X when $U = u_1u_2u_3 \ldots u_k$, having $u_l = x_{i_l}$ and $i_l < i_{l+1}$ for all l with $1 \leq l < k$. Given two strings X and Y, a longest common subsequence (LCS) of both strings is defined as any string which is a subsequence of both X and Y and has maximum possible length. We consider the problem of finding the length of such a sequence, which will be denoted as LLCS(X, Y).

The basic dynamic programming algorithm for this problem defines the dynamic programming matrix $L_{0\ldots m,0\ldots n}$ as follows (see also [41, 15]):

$$L_{i,j} = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ L_{i-1,j-1} + 1 & \text{if } x_i = y_j, \\ max(L_{i-1,j}, L_{i,j-1}) & \text{if } x_i \neq y_j \ . \end{cases} \tag{5.1}$$

The value $L_{i,j}$ is equal to LLCS($x_1x_2 \ldots x_i$, $y_1y_2 \ldots y_j$). Using dynamic programming [15], the values in this matrix can be computed in $O(mn)$ time and space. When computing L row by row, we can reduce the space requirement to $O(n)$, as we only need to store the current and previous rows $L_{i-1,\cdot}$ and $L_{i,\cdot}$ at any given moment. Figure 5.1(a) shows an example for the dynamic programming matrix, and Figure 5.3 shows the algorithm used for local computation.

The problem of computing the LLCS is equivalent to finding the length of the longest path from $(0, 0)$ to (m, n) in a grid directed acyclic graph as in Figure 5.1(b). This graph has the vertices $V = \{(i, j) \mid 0 \leq i \leq m \text{ and } 0 \leq j \leq n\}$. All vertical edges $(i, j-1) \rightarrow (i, j)$

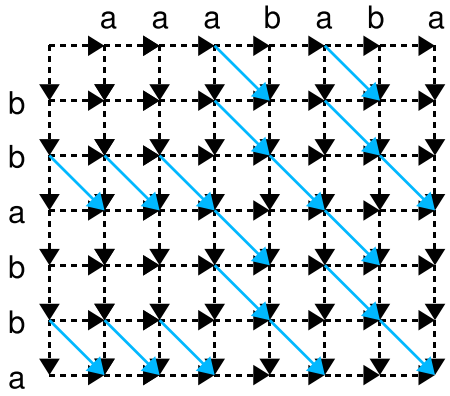
and horizontal edges $(i-1,j) \rightarrow (i,j)$ have weight 0. The diagonal edges $(i-1,j-1) \rightarrow (i,j)$ have weight 1 if $x_i = y_j$ (shown in blue); otherwise their weight is 0. The graph representation shows data dependencies more clearly: to compute each element $L_{i,j}$ with $0 < i \leq m$ and $0 < j \leq n$, we need the values of $L_{i-1,j}$, $L_{i,j-1}$ and $L_{i-1,j-1}$.

For deriving a BSP algorithm, we use a simple parallel algorithm for grid DAG computation. The matrix L is partitioned into a grid of rectangular blocks with size $(m/G) \times (n/G)$, where the parameter G specifies the grid size. This enables us to compute L in $2G-1$ parallel steps. The data in all blocks on the wavefront (shown in dark blue in Figure 5.2(a)) only depends on the data in the blocks of the previous wavefront (shown in light blue). Therefore, these blocks can be processed in parallel. Figure 5.2(b) shows that this process takes 9 parallel steps on 3 processors when $G = 5$. The parallel algorithm is shown in Figure 5.4. Array B of length $m+1$ contains the values $L(0,\dots)$ for each local block. Hence, input/output values in B can be kept locally. They correspond to the data that is highlighted green in Figure 5.2(a). The corresponding array R of length $n+1$ containing $L(\dots,0)$ has to be transferred to the processor that is assigned the computation of the block to the right of the current block in the block grid. This communication pattern is equivalent to a cyclic shift. After running the parallel algorithm, processor G mod p holds the part of L which contains the LLCS.

The algorithm shown in Figure 5.4 requires $(2G-1) \cdot G/p$ supersteps, assuming the ratio

		a	a	a	b	a	b	a
	0	0	0	0	0	0	0	0
b	0	0	0	0	1	1	1	1
b	0	0	0	0	1	1	2	2
a	0	1	1	1	1	2	2	3
b	0	1	1	1	2	2	3	3
b	0	1	1	1	2	2	3	3
a	0	1	2	2	2	3	3	4

(a) Dynamic programming matrix



(b) Grid DAG

Figure 5.1: LLCS dynamic programming approach for the strings *aaababa* and *bbabba*

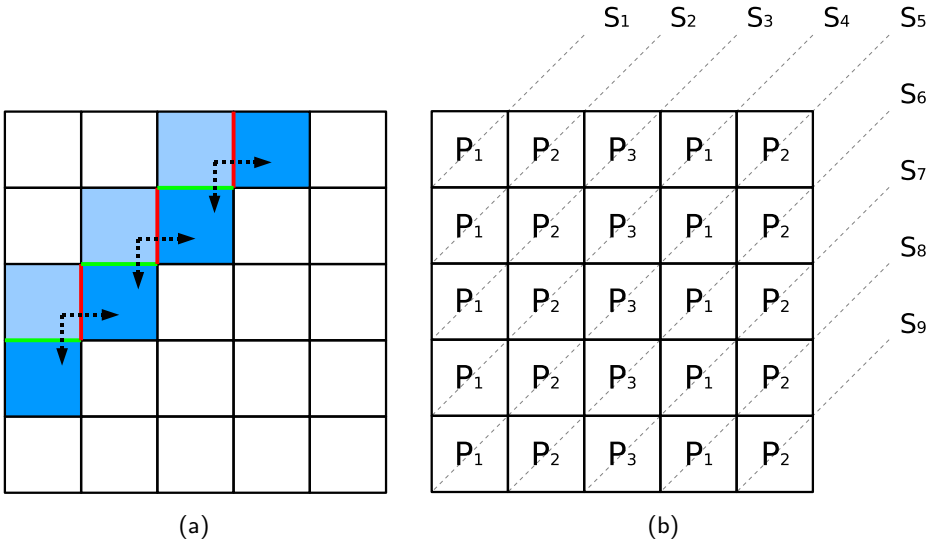


Figure 5.2: Parallel LLCS: blocked wavefront approach for $p = 3$ and $G = 5$

$\alpha = G/p$ to be an integer. This simplifies the following performance analysis. Moreover, there is no performance advantage in considering the case where G is not a perfect multiple of p. Between the supersteps, we need to transfer the rightmost column and bottom row of the local part of L (highlighted in red and green in Figure 5.2(a)). When subproblem blocks are assigned to processors block-cyclically as in Figure 5.2(b), the values in the bottom row and the corresponding part of string X can be kept locally and do not have to be transferred using the communication network, because one processor is always assigned blocks from the same column. For simplicity, we assume that the strings have equal length $n = m$ and that the input data distribution is block-cyclic with blocks of size $\lceil n/G \rceil$.

In estimating the computation time, we need to consider that in general there are less than G/p blocks in every wavefront. The number of blocks only equals G/p when the wavefront reaches the diagonal of the block grid. Still assuming that $G = \alpha p$ with an integer α, we

Require: X, Y are the input sequences, B contains the values $L(0,\dots)$ and has length $m+1$, R contains $L(\dots,0)$ and has length $n+1$

initialize pcl to point to an array of $m+1$ integers
initialize ppl to point to B
for $j = 1$ **to** n **do**
  $pcl(0) \leftarrow R(j)$
  for $k = 1$ **to** m **do**
    if $x_k = y_j$ **then**
      $pcl(k) \leftarrow ppl(k-1) + 1$
    else
      $pcl(k) \leftarrow max(ppl(k), pcl(k-1))$
    end if
  end for
  $R(j) \leftarrow pcl(m+1)$
  Swap pointers pcl and ppl
end for
Set $B = ppl$ if necessary
return (B, R) $\{B(m+1)$ contains the LLCS$\}$

Figure 5.3: Algorithm LLCS(X,Y,B,R)

get

$$
\begin{aligned}
W &= \left\lceil \frac{n}{G} \right\rceil^2 \cdot \left(\left\lceil \frac{1}{p} \right\rceil + \left\lceil \frac{2}{p} \right\rceil + \cdots + \left\lceil \frac{G}{p} \right\rceil + \left\lceil \frac{G-1}{p} \right\rceil + \left\lceil \frac{G-2}{p} \right\rceil + \cdots + \left\lceil \frac{1}{p} \right\rceil \right) \\
&= \left\lceil \frac{n}{G} \right\rceil^2 \cdot \left(-\left\lceil \frac{G}{p} \right\rceil + 2\sum_{j=1}^{G} \left\lceil \frac{j}{p} \right\rceil \right) \quad = \quad \left\lceil \frac{n}{G} \right\rceil^2 \cdot \left(-\left\lceil \frac{G}{p} \right\rceil + 2\sum_{j=1}^{\alpha} pj \right) \\
&= \left\lceil \frac{n}{G} \right\rceil^2 \cdot (G^2/p + G - G/p) \quad = \quad \left\lceil \frac{n}{\alpha p} \right\rceil^2 \cdot (p\alpha(\alpha+1) - \alpha) \ . \qquad (5.2)
\end{aligned}
$$

The theoretical behavior of the computation cost as a function of α is shown in Figure 5.5. The asymptotic cost decreases for $p > 1$ and increasing α, but becomes unstable when n is not a perfect multiple of αp. Input data and results have to be transferred for $G(G-1)$ blocks (results have to be transferred for all blocks in the grid not located at the right border, and input data for the second string has to be fetched for all blocks not on the diagonal). Individual characters are stored in one byte (thus $\log_2 \sigma = 8$), and 4-byte/32-bit integer values represent the dynamic programming matrix elements. Each processor sends a column of matrix elements and receives a part of the second input string, which is distributed block-

Require: X, Y are the input sequences, p contains the number of processors, pid the current processor (counting from 1) and $G = \alpha p$ the grid size.

Allocate arrays $\mathrm{R_j}$ of size $\lceil n/G \rceil + 1$ and $\mathrm{B_j}$ of size $\lceil m/G \rceil + 1$ for $j = 1, \ldots, \lceil G/p \rceil$
for $diag = 1$ **to** $2G - 1$ **do**
  for $block = 1$ **to** G/p **do**
    $x \leftarrow (block - 1) \cdot p + pid$
    $y \leftarrow diag - x + 1$
    Compute local block width $bw \approx m/G$ and height $bh \approx n/G$
    if (x, y) is inside the grid $(1, 1), \ldots, (1, G), (2, G), \ldots, (G, G)$ **then**
      Get the parts X_p and Y_p of X and Y that correspond to the dynamic programming subproblem at position (x, y)
      Synchronize
      Call $\mathrm{LLCS}(X_p, Y_p, B_{block}, R_{block})$
      if $x = G$ **and** $y = G$ **then**
        $llcs \leftarrow B_{block}(bw)$
      end if
      Put R_{block} to processor $(pid + 1)$ mod p
    else
      Synchronize
    end if
  end for
end for
return $llcs$ {On processor G mod p, broadcast if necessary}

Figure 5.4: Algorithm $\mathrm{PAR\_LLCS}(X,Y)$

cyclically[1]. We further introduce a factor of $1/64$, as g was measured in 64-bit double/s. Hence we have to multiply the theoretical communication cost with this factor to obtain the practical value:

$$\frac{\overbrace{32}^{h^{out}} + \overbrace{\log_2 \sigma}^{h^{in}}}{64} = \frac{5}{8} .$$

Altogether, the BSP running time is

$$\begin{aligned} T(\alpha) = \quad & f \cdot (p\alpha(\alpha+1) - \alpha) \cdot \left\lceil \frac{n}{\alpha p} \right\rceil^2 \\ + \quad & g \cdot \frac{5}{8} \cdot \alpha(\alpha p - 1) \left\lceil \frac{n}{\alpha p} \right\rceil \\ + \quad & l \cdot (2\alpha p - 1) \cdot \alpha \ . \end{aligned} \tag{5.3}$$

[1]Another method would be to broadcast the second input string to all processors at the beginning. This is not done here to preserve memory-efficiency when working on long strings of equal length.

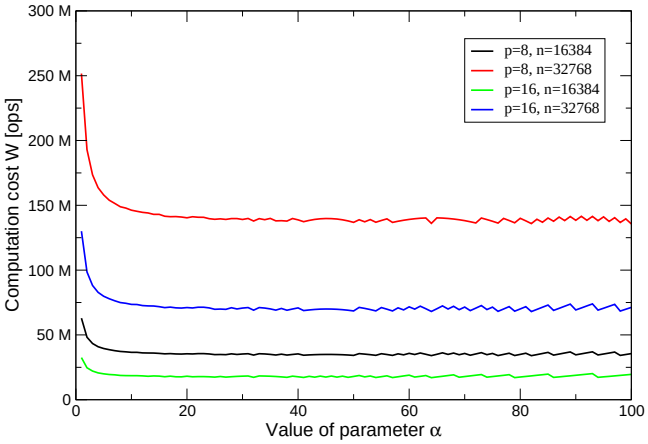


Figure 5.5: LLCS computation — Dependency of W on grid size factor

The parameter α can be used to tune the performance of the algorithm. Knowing a minimum block size b for which the sequential algorithm achieves good performance (e.g. the processor's cache size), the critical value of α can be pre-calculated before computation:

$$\alpha = \frac{5 \cdot n}{p \cdot b} \ . \tag{5.4}$$

This choice of α ensures that all the data necessary for computing one row of a dynamic programming matrix block can be stored in a block of size b. In the experiments, this method established good balance between having to minimize overhead from partitioning small problems into too many blocks, and achieving higher computation speed when the data for the local computations can be stored in the CPU cache.

5.2 Bit-Parallel Algorithm

Computing the LLCS using the standard dynamic programming approach is by far not the fastest method available. Let w be the bit-length of a machine word. Assuming that both integer and bitwise Boolean operations are available, the sequential computation time can be reduced by an approximate factor of $1/w$ using algorithms as proposed by Crochemore et

Operator	Notation	Example
Bitwise-not (or one's complement)	$\sim$	$\sim 1101 = 0010$
Bitwise inclusive-or	$\mid$	$1101 \mid 1010 = 1111$
Bitwise-and	$\&$	$0010 \ \& \ 1010 = 0010$
Addition with carry	$+$	$1101 + 0011 = 0001$; carry $= 1$

Table 5.1: Bit operators in C notation

al. [18] or Allison and Dix [2] (see the beginning of this Chapter for more references to related literature). The basic idea of these algorithms is to work with the differences in the dynamic programming matrix $\Delta L(i,j) = L(i,j) - L(i-1,j)$. These differences can have either the values 0 or 1, and thus be encoded as a bit string. The different bit-parallel algorithm variants have a general structure in common: they compute $\Delta L(i,j)$ as a function of $\Delta L(i-1,j)$ and a mapping $M(x)$, which maps a character x to a bit string of length n (i.e. the height of the dynamic programming table). Performance is gained because the values of $\Delta L(i,j)$ can be computed using integer and bitwise Boolean operations, which work on w bits in parallel. Our implementation uses the algorithm from [18]. The other variants differ from this one in the number of operations used in the recurrence. However, all variants include integer operations that can create a carry. This keeps the data dependence pattern of the basic dynamic programming algorithm and makes the same wavefront approach necessary that was used in the previous section. Different methods for implementing the mapping $M(x)$ are discussed in [46]. We use the method of storing all the bit-vectors in an array of size $O(n \cdot \sigma)$, which is simple and fast, but also memory inefficient.

```
unsigned long add_with_carry(
    unsigned long a,
    unsigned long b,
    unsigned long & c) {
    unsigned long r= a + b + c;
    c= (     (a > ULONG_MAX - c)
         ||  (a + c > ULONG_MAX - b)
       ) ? 1 : 0 ;
    return r;
}
```

Figure 5.6: Implementing addition with carry in C++

To denote the bitwise Boolean operations *not*, *and* and *or*, we use the same notation as

in the C programming language (see Table 5.1). Figure 5.7 shows the sequential algorithm in more detail. It outputs not the LLCS, but the carry values B and the values of $R = \sim \Delta L(m)$. It gets values B and R as input, which are initialized with $B \leftarrow 0$ and $R \leftarrow 2^n - 1$. Moreover, we introduce the operation of an addition with carry, a C++ implementation of this operation is shown in Figure 5.6[2]. After running the sequential algorithm, R contains the values of $\sim \Delta L(m, \dots)$. The LLCS can then be obtained by counting the number of zeros in R, as is proven in [18].

Require: X, Y are the input sequences, B is an array of carry bits of length m, R contains $\sim \Delta L(0)$

1: $\forall \gamma \in \Sigma . M(\gamma) \leftarrow 0$
2: **for** $j = 1$ **to** n **do**
3: $\quad M(y_j) \leftarrow M(y_j) \mid 2^{j-1}$
4: **end for**
5: **for** $k = 1$ **to** m **do**
6: $\quad R \leftarrow (B(k) + R + (R \;\&\; M(x_k))) \mid (R \;\&\; (\sim M(x_k)))$
7: $\quad B(k) \leftarrow$ carry
8: **end for**
9: **return** (R, B)

Figure 5.7: Algorithm $\mathrm{BITPAR\_LLCS}(X,Y,R,B)$

The parallel version uses the same dynamic programming scheme as the simple algorithm in Section 5.1. In our wavefront approach from Figure 5.2(a), the carry values B will be passed downwards (green) and the parts of R to the right (red). To obtain the LLCS, the zero bits in all parts of R at the right side of the block grid are counted and summed up. This can be done in one additional superstep. We introduce a modified asymptotic computation speed f', which will have a value of approximately $\frac{1}{w} f$ (see Table 5.4 on page 78 for practical speedups). In our implementation, the asymptotic computation speed is obtained for larger problem sizes compared to the standard sequential LLCS algorithm, because computing the mapping table M (see lines 1 and 2 in Figure 5.7) introduces a constant amount of overhead that depends on the alphabet size. Furthermore, our method of storing M completely consumes an amount of memory that is exponential in the alphabet size. For alphabets

[2] This implementation is likely to be compiled to code which includes branching instructions on some systems and includes redundant additions and comparisons to retrieve the carry. For optimal performance, it should be replaced by assembly code, this usually allows direct carry retrieval.

larger or equal to 2^{16} (the alphabet size for the Unicode alphabet), match vector retrieval should be implemented differently [46]. However, for the experiments with the bit-parallel algorithm, we use an alphabet size of 8 characters (i.e. 3 bits per character). Therefore, the simple method is sufficient. The BSP running time for the parallel dynamic programming algorithm using bit-parallel computation is obtained similarly as described in the previous Section. Having a communication cost factor of

$$\frac{1 + \log_2 \sigma}{64} = \frac{4}{64} = \frac{1}{16},$$

we get:

$$\begin{aligned} T(\alpha) = \quad & f' \cdot (p\alpha(\alpha+1) - \alpha) \cdot \left\lceil \frac{n}{\alpha p} \right\rceil^2 \\ + \quad & g \cdot \frac{1}{16} \cdot \alpha(\alpha p - 1) \left\lceil \frac{n}{\alpha p} \right\rceil \\ + \quad & l \cdot ((2\alpha p - 1) \cdot \alpha + 1) \ . \end{aligned} \tag{5.5}$$

5.3 Experiments for the Simple Algorithm

The experiments for the LLCS algorithms were carried out on the same systems that were introduced in Section 2.4. The alphabet size σ was set to 8 characters. Each character is stored in one byte to avoid overhead for data access. The dynamic programming table used 32-bit integer values. Input strings were generated randomly and had equal length. The first

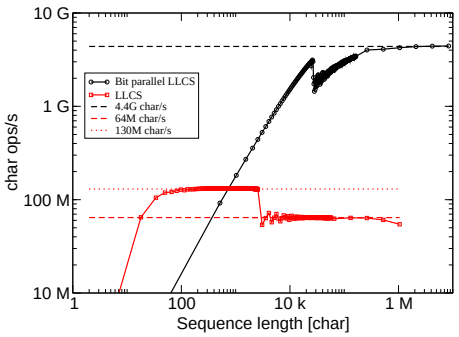
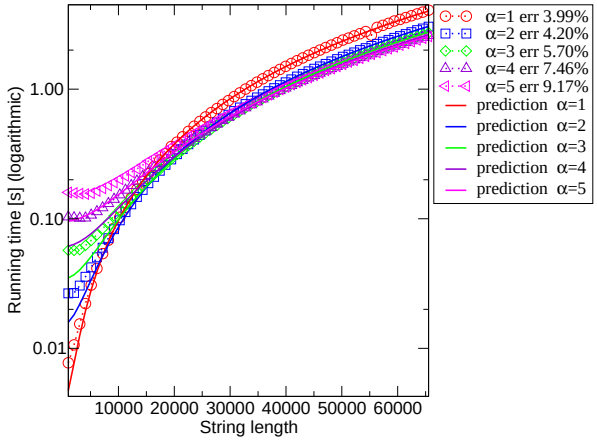


Figure 5.8: Character rate F (`skua`)

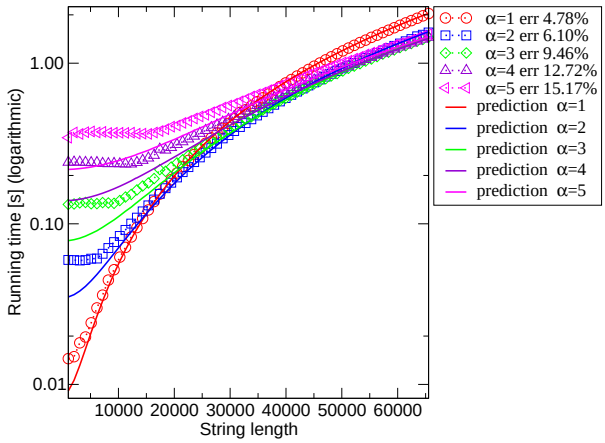
	Simple Algorithm	
`skua`	0.008 ns/op	130 M op/s
`argus`	0.016 ns/op	61 M op/s
`aracari`	0.012 ns/op	86 M op/s

	Bit-Parallel Algorithm	
`skua`	0.00022 ns/op	4.5 G op/s
`argus`	0.00034 ns/op	2.9 G op/s
`aracari`	0.00055 ns/op	1.8 G op/s

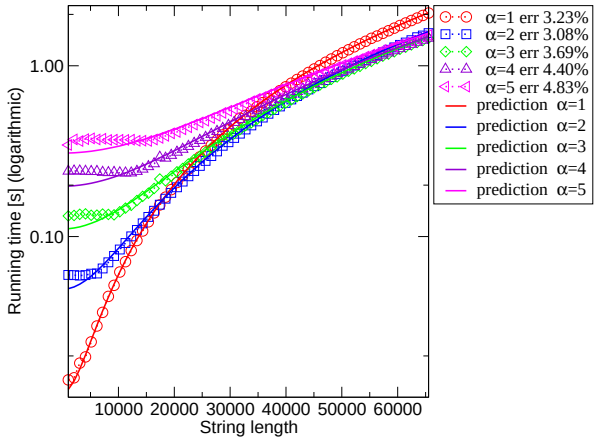
Table 5.3: Experimental values of f



(a) MPI, 16 processors, $g = g(h)$



(b) Oxtool, 32 processors, g constant



(c) Oxtool, 32 processors, $g = g(h)$

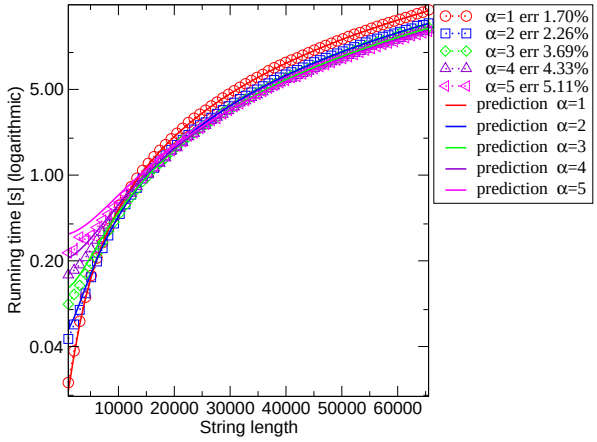
Figure 5.9: LLCS (small) — Predictions on `skua`

set of experiments was run using string lengths between 8192 and 65536, and values of the grid size parameter α between 1 and 5, to study the predictability for small problem sizes. Since we use the `put` primitive, which has low per-message overhead, using the variable bandwidth model $g = g(h, h^*)$ does not improve prediction accuracy. Furthermore, there is only one `put` request per superstep. However, using $g = g(h)$ reduces the prediction error for small problem sizes. The values of l, g and h_{half} measured for conducting random permutations using the `put` primitive were used to create the predictions, as this best matches the implementation's communication pattern. Table 5.3 shows the values of f used for prediction. As in the previous Chapter, they were obtained by measuring local computation times for one run.

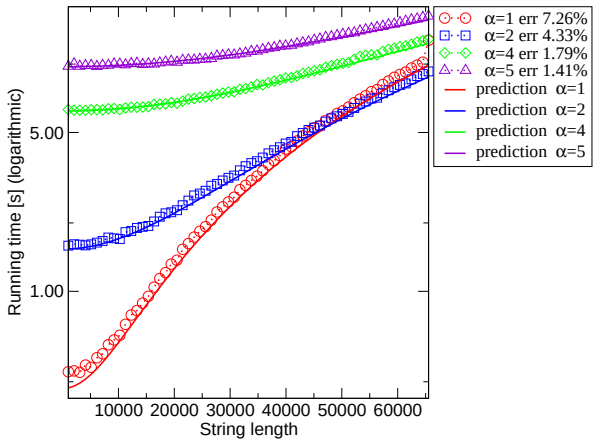
Altogether, the prediction results are much better than the ones obtained for matrix multiplication. The first reason for this is that communication cost only increases linearly with the problem size. Hence, the performance model is less sensitive to fluctuations in the values of g and l than the model for matrix multiplication, in which communication costs increase quadratically with the input size. Furthermore, the communication pattern of the LLCS algorithm is simpler and more regular. Moreover, the largest part of the data is exchanged using `put` requests that achieve more predictable performance on all systems. A complete listing of results can be found in Appendix F.

Results on Shared Memory (`skua`)

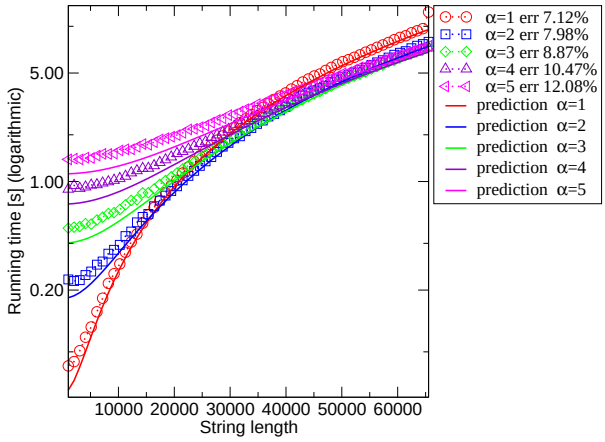
On `skua`, the predictions for Oxtool and MPI matched the experimental results very well (see Figure 5.9). The predicted running times were lower than the measurements using PUB, the same behavior was observed for the matrix multiplication algorithm. Extending the model by using h_{half} improves the prediction quality on this system. This can be seen clearly when the number of processors is high and the communication size is small. Good efficiency of up to 80% is achieved by Oxtool and MPI. PUB only achieves approximately 60% because its effective communication bandwidth is lower.



(a) Oxtool, 4 processors, g constant



(b) MPI, 10 processors, g constant



(c) PUB, 10 processors, $g = g(h)$

Figure 5.10: LLCS (small) — Predictions on `argus`

Results on Distributed Memory, Ethernet (argus)

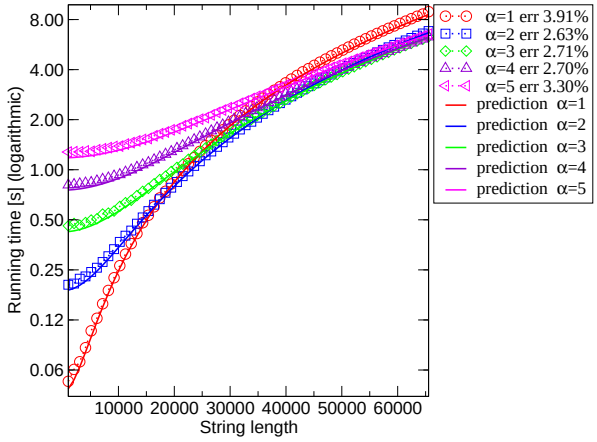
On argus, the best predictability using a constant value of g is achieved by Oxtool due to its low synchronization latency. The prediction results for PUB and MPI can be improved and made similar or better compared to Oxtool by using $g = g(h)$ (see Figure 5.10). The advantages of the optimized libraries for small messages can be seen particularly well on this system: the run time for the MPI version improves drastically when using smaller values of the grid size parameter α. When α is larger, the dynamic programming algorithm particularly benefits from a low synchronization latency. MPI has very high latency, and thus produces long running times even for small problem sizes. Best efficiency and superlinear speedup on 4 processors is achieved by Oxtool and PUB with slight advantages for Oxtool. MPI cannot benefit from the lower computational costs for higher values of α, as these are compensated by higher synchronization costs.

Results on Distributed Memory, Myrinet (aracari)

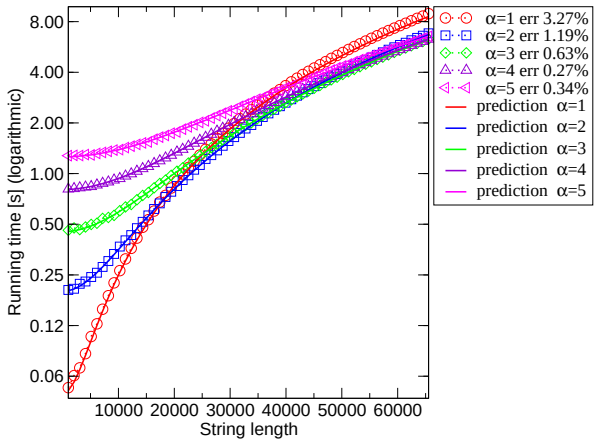
On aracari, a larger number of processors leads to greater prediction errors when using $g = g(h)$ to predict the running time for PUB or Oxtool. Obviously, the effective overhead for small communication sizes is lower than the value of h_{half} predicts. The best predictability is achieved by MPI. When using MPI and $g = g(h)$, the mean relative prediction error is smaller than 5% for all numbers of processors. Best efficiency is achieved by PUB and Oxtool for similar reasons as on argus. PUB benefits from having a low synchronization latency, Oxtool has the better effective bandwidth. On average, both achieve similar performance. The performance for MPI drops when the number of processors is larger, as its synchronization latency becomes higher.

Speedup for Larger Problems

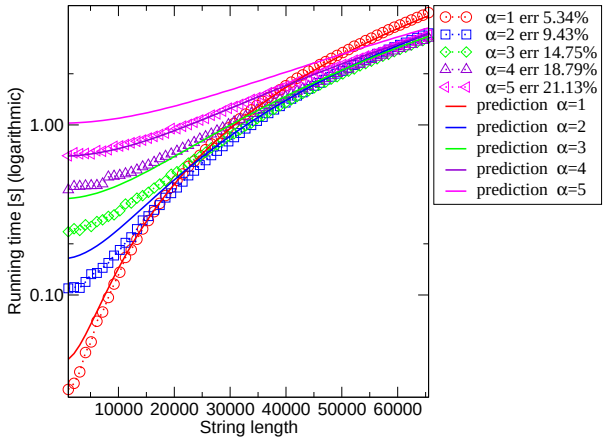
The performance of the algorithm was evaluated by running it on larger problems and using an optimized value of α from Equation (5.4). The speedup results are shown in Figure 5.12. It can be seen that there are few differences between MPI, PUB and Oxtool for large problems. For short strings, Oxtool and PUB show advantages over MPI on the message passing systems (aracari and argus), particularly when the number of processors is large. On



(a) MPI, 16 processors, g constant

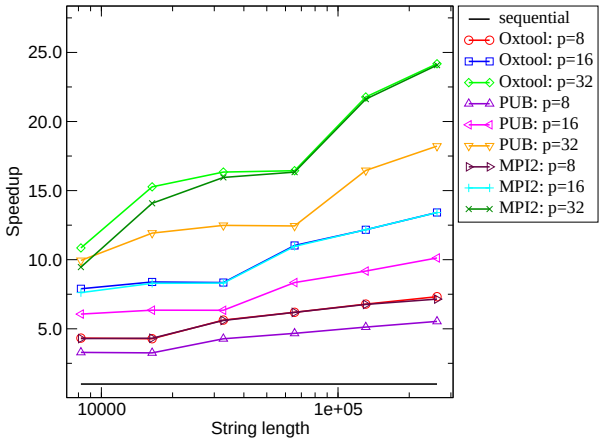


(b) MPI, 16 processors, $g = g(h)$

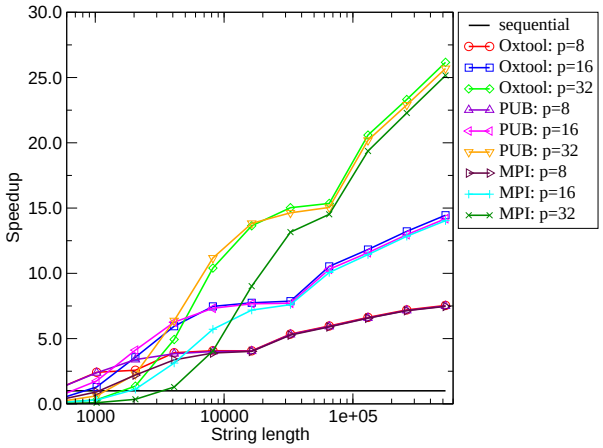


(c) PUB, 32 processors, $g = g(h)$

Figure 5.11: LLCS (small) — Predictions on `aracari`



(a) Speedup on `skua`



(b) Speedup on `aracari`

Figure 5.12: LLCS computation — Speedup

skua, PUB obviously has difficulties when the communication cost is too large, and achieves substantially lower speedup than MPI or Oxtool. The speedup graphs also demonstrate well that MPI only achieves comparable speedup to an optimized communication library if the problem size is large.

5.4 Experiments for the Bit-Parallel Algorithm

The bit-parallel algorithm shows much better local computation performance (see Table 5.4) on all systems and also has lower communication costs. However, the asymptotic computation speed on skua is only reached for very large problem sizes (see Figure 5.14). This system uses 64-bit machine words, which allows a higher degree of bit-parallelism, that obviously is only exploited gradually with increasing problem size by our high-level programming language implementation on this system. To explore this further, tests were run on another 64-bit machine, a desktop system named gnoll, which has an AMD Athlon64 processor. On this system, the software was compiled using the GNU C Compiler [77]. Figure 5.14 shows that on this same system the bit-parallel variant reaches its peak performance only for larger problems than the standard algorithm. However, the effect is not as visible as on skua, and no cache threshold point can be seen. On the other systems (aracari and argus), the behavior was similar to gnoll (although having a lower speedup, as these systems only use 32-bit words). This leads to the conclusion that these effects are specific to the Itanium-2

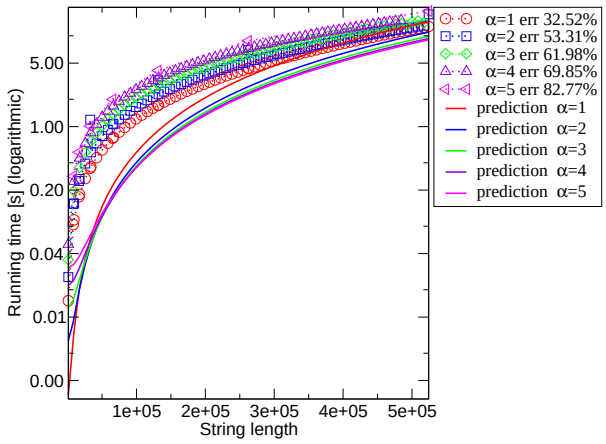
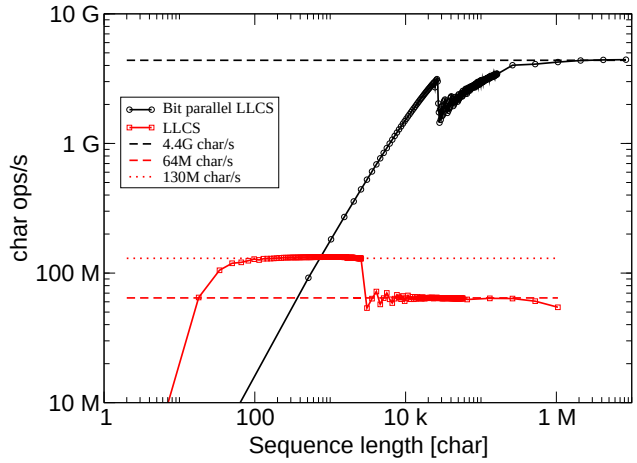
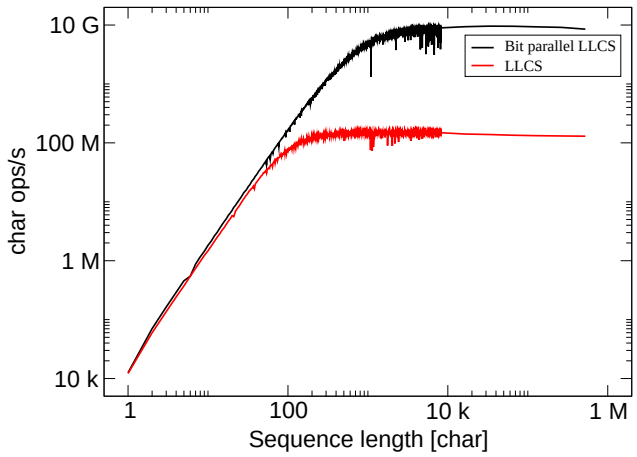


Figure 5.13: Bit-Parallel LLCS — Predictions on skua (MPI, using 8 processors)

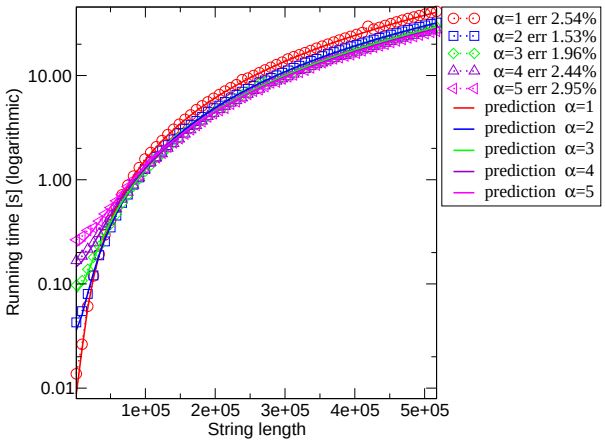


(a) `skua`

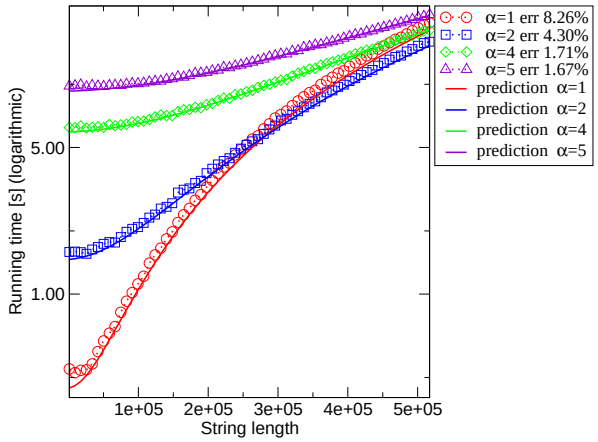


(b) `gnoll`

Figure 5.14: Bit-parallel LLCS computation using 64-bit integers



(a) `aracari`, PUB, 8 processors, $g = g(h)$



(b) `aracari`, Oxtool, 8 processors, $g = g(h)$

Figure 5.15: Bit-Parallel LLCS — Predictions on `aracari`

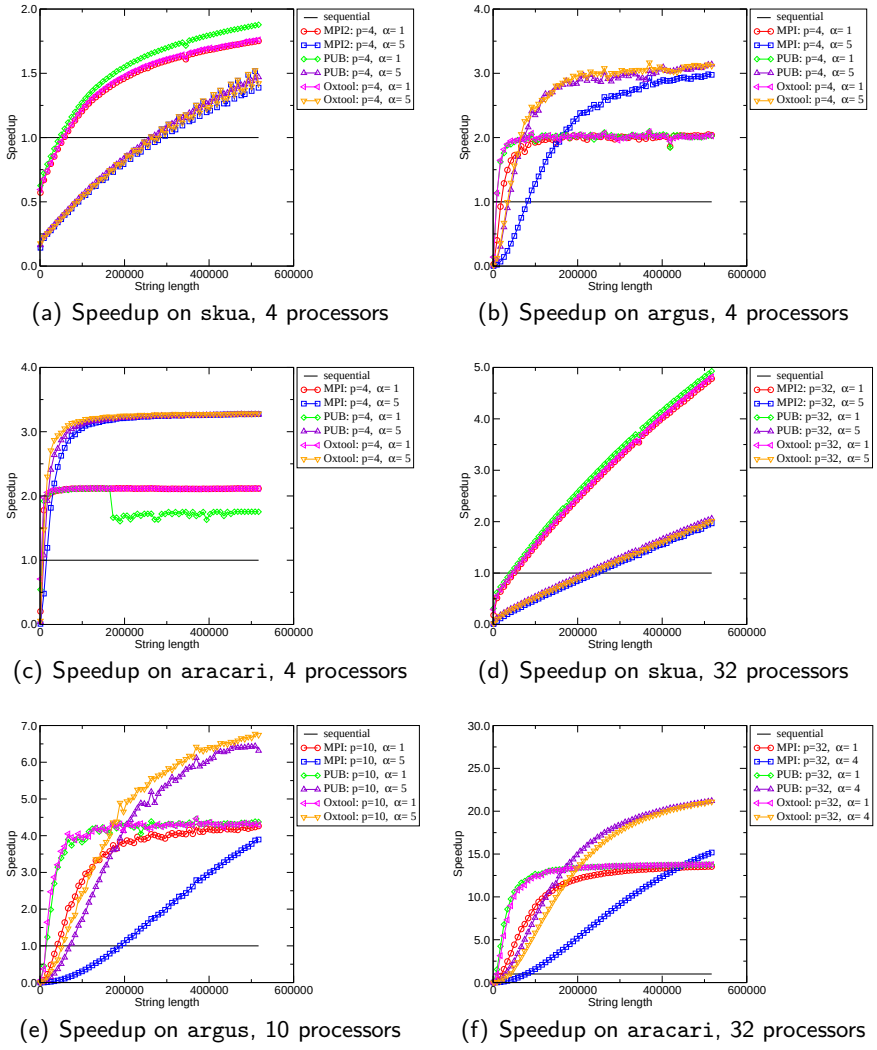
System	Sequential computation speedup	
skua	34.6 (70.3)	(130 M → 4.5 G char-op/s)
gnoll	63.3	(150 M → 9.5 G char-op/s)
argus	47.5	(61 M → 2.9 G char-op/s)
aracari	20.9	(86 M → 1.8 G char-op/s)

Table 5.4: Sequential computation speedup from using bit-parallel computation

System (skua) in combination with the Intel C++ compiler [79]. For calculating the sequential computation speedup compared to the standard algorithm on skua, we assume that the standard algorithm uses block partitioning to work with problem sizes, for which cache efficiency is achieved (this can be done using an optimized grid size on one processor, see Equation (5.4) on page 65). Therefore, only a speedup factor of approximately 35 is obtained, instead of 64 from using 64-bit-parallel operations. When comparing the asymptotic character rate for the standard algorithm to the bit-parallel variant, a speedup factor of 70.3 is obtained.

Supplying the sequential computation code in assembly instead of C++ is likely to improve the performance predictability for this algorithm on skua. However, we still used the C++ code to ensure portability. As the performance for the problem sizes that arise in our experiments is not constant using this code (see the discussion above), running times on skua cannot be predicted accurately using a constant value of f. Figure 5.13 shows that the predicted running times are smaller than the measured times, especially when the block size decreases if α is larger or more processors are used. On the other systems, the predictions were of similar quality as the ones made for the simple LLCS algorithm. Figure 5.15 shows selected prediction plots. On aracari, PUB shows the same performance drop as before: when the communication cost becomes larger than approximately 2048 bytes, performance decreases.

Figure 5.16 shows the speedup results. On all systems, the speedup achieved when using the bit-parallel sequential algorithm is lower compared to the experiments using the standard algorithm (however, the running time using the bit-parallel algorithm is always lower than using the standard algorithm). On the shared memory machine, this is caused by a lower sequential computation speed for small subproblem sizes (as discussed above). An increased grid size leads to lower sequential computation performance, as this decreases the subproblem



(a) Speedup on `skua`, 4 processors

(b) Speedup on `argus`, 4 processors

(c) Speedup on `aracari`, 4 processors

(d) Speedup on `skua`, 32 processors

(e) Speedup on `argus`, 10 processors

(f) Speedup on `aracari`, 32 processors

Figure 5.16: Bit-Parallel LLCS — Speedup

size. However, further increasing the problem size on the shared memory system presumably would yield better speedup.

On the distributed memory systems, the parallel speedup when using bit-parallel computation instead of the standard algorithm is also lower. Still, good speedup is obtained both on Myrinet (`aracari`) and Ethernet (`argus`). The best performance on `argus` is achieved by Oxtool, and the performance of PUB is only slightly worse. On 10 processors, the difference between plain MPI and the optimized libraries is most visible. When using higher values of α, PUB and Oxtool achieve better speedup, whereas MPI shows decreasing performance due to its high overhead for small communication costs. When using 4 processors on `aracari`, the performance drop of PUB, starting at a certain communication cost, is clearly visible (see Subfigure 5.16(c)). When the number of processors and hence the block grid size increases, the communication cost becomes lower and PUB shows very good performance. When using a large grid size on `aracari`, PUB outperforms both MPI and Oxtool.

Chapter 6

Conclusion

In this work we studied the performance of bulk-synchronous parallel algorithms using optimized special purpose communication libraries. We described extensions to the standard BSP model that can improve performance predictions, and proposed a mechanism for benchmarking that can reliably be used to obtain standard and advanced BSP parameters. Furthermore, we conducted experiments for two different kinds of algorithms, matrix multiplication and longest common subsequence computation, in order to evaluate the practical performance of BSP algorithms on different parallel computers. In particular, we tried to use efficient implementations for sequential computation to achieve comparability of our results with other optimized implementations. As BSP also has uses as a performance model, methods of performance prediction were investigated and their accuracy studied.

6.1 Result Summary

Experiments were conducted on three different parallel machines: a shared memory system and two PC clusters, one with an Ethernet, the other with a Myrinet interconnection network. We compared the performance of different communication libraries: PUB, the Oxford BSP Toolset, and a simple BSP library based on MPI. PUB and the Oxford BSP Toolset both implement optimizations to improve communication performance.

The first set of experiments was concerned with measuring the communication and computation performance separately to obtain parameters for describing the parallel machines

using the BSP model, particularly focusing on comparing parameters that describe the performance of the communication network. Communication benchmarks showed that there can be a large difference in performance between different libraries. On the message passing systems, there were advantages for the optimized libraries. The Oxford BSP Toolset achieved particularly good performance on both Ethernet and Myrinet. PUB only showed good performance on the Ethernet system. On Myrinet, having a higher communication cost increased the per-message overhead when using PUB, hence only sub-optimal performance and predictability could be obtained. MPI was not able to deliver comparable performance either to PUB or the Oxford BSP Toolset on the Ethernet system. On Myrinet, MPI outperformed PUB for large communication costs, but could not achieve the same communication performance as the Oxford BSP Toolset. On the shared memory machine, the optimized BSP libraries could only achieve better performance than MPI for small messages using message passing or remote memory `put` primitives. In contrast, the remote memory read access using the MPI-2 DRMA functionality achieved the best communication performance on this system.

Further experiments were carried out using memory-efficient parallel matrix multiplication. The results from these experiments showed that even simple BSP algorithm implementations can achieve similar or better speedup compared to PBLAS, an optimized standard library. The only exception was the Ethernet system, on which comparable performance to PBLAS could only be obtained for small numbers of processors, as our memory efficient algorithm uses a communication pattern that is not efficient on this system. However, our implementation was able to outperform PBLAS on Myrinet using the Oxford BSP Toolset, and on shared memory using MPI-2. The performance could be predicted with reasonable accuracy by using extensions to the BSP model that describe overhead for small messages and per-message overhead. However, as this algorithm's communication cost increases quadratically with the problem size, the performance model showed particular sensitivity to errors in the communication time estimation. This complicates accurate prediction, especially when using a parallel machine with varying traffic on a communication network that is shared among many users.

Our second practical benchmark used a parallel dynamic programming algorithm that

computes the length of the longest common subsequence of two strings. This algorithm has linear communication cost and quadratic computation cost. Performance predictability was very good. However, when using an efficient sequential implementation, good speedup could only be obtained for very large problem sizes. The PUB implementation of this algorithm showed very good performance, as the communication cost is small and PUB has advantages when each superstep consists of a small number of messages.

Table 6.1 shows a summary of all results. The number of bullets specifies the performance that was achieved by each library, ranging from • (lowest performance compared to the other libraries) to ••• (best performance). If the results were close, both the best/worst libraries were awarded •••/•.

Table 6.1: Experimental result summary

Experiments	**Oxtool**	**PUB**	**MPI**
Shared memory (`skua`)			
Communication gap (`put`/`get`)	••	•	•••
Communication gap (`send`)	•••	••	•
Latency	•••	•	•
Matrix multiplication	••	•	•••
LLCS (standard)	•••	•	••
LLCS (bit-parallel)	••	•••	•
Distributed memory, Ethernet (`argus`)			
Communication gap (`put`/`get`)	•••	•••	•
Communication gap (`send`)	••	•••	•
Latency	•••	••	•
Matrix multiplication	•••	••	•
LLCS (standard)	•••	••	•
LLCS (bit-parallel)	•••	••	•
Distributed memory, Myrinet (`aracari`)			
Communication gap (`put`/`get`)	•••	•	••

Continued on the next page...

...continued from last page.

Experiments	Oxtool	PUB	MPI
Communication gap (`send`)	●●●	●	●●
Latency	●●	●●●	●
Matrix multiplication	●●●	●	●●
LLCS (standard)	●●●	●●	●
LLCS (bit-parallel)	●●	●●○[1]	●

6.2 Outlook

The communication structure of BSP-like algorithms bears potential for optimizing performance at library level. The experiments on state of the art systems showed that optimized BSP libraries can achieve better performance than a plain general-purpose communication library, though this is not always the case. Both PUB and Oxtool have communication devices for several different architectures, and compiling them on top of MPI usually does not yield the best possible performance. On the other hand, our experiments have shown that performance improvement is often possible, even when using MPI as a basis for a BSP library. Hence, an optimized BSP framework using MPI or any other general purpose communication library could achieve performance portability also to future parallel systems.

Furthermore, all our experiments were focused on homogeneous parallel computers that use a fast communication network. However, there are also various projects that aim to merge the BSP approach and the idea of Grid computing [25], or to run BSP-style algorithms on heterogeneous parallel computers. A BSP based model for heterogeneous parallel computing is proposed in [53]; examples of BSP programming frameworks for the Grid or similar parallel architectures are InteGrade [31], GridNestStep [54] and the PUB Web-computing Library [11]. This involves a whole range of interesting new problems, such as efficient job scheduling, fault-tolerance and above all, obtaining run-time efficiency on systems which may have varying processor-to-processor communication speed, as well as different memory

[1] Best performance until drop at $h = 2048\ bytes$.

and processing capabilities. Having a simple, efficient and portable framework for BSP programming on heterogeneous computers would enable BSP algorithms to make use of the great computational power that is available in the Grid. On the other hand, Grid computing is likely to benefit from implementations that have predictable performance, as this allows more efficient job scheduling and usage of computational resources.

The programming model we used is based on the BSPlib standard. This can complicate performance prediction, as `put` and `get` primitives may have different performance. Also, different BSP libraries obviously have been optimized for different kinds of communication. The Oxford BSP Toolset shows very good performance using `put`-like communication, PUB shows better performance when using `send`/`move`. This problem could presumably be avoided by using a more abstract programming model similar to CGMlib's h-relation [69] or SSCRAP's bulk communication functionality [24]. Furthermore, having a more abstract programming model would facilitate portable library-level optimization. Another possibility is to combine the idea of algorithmic skeletons and BSP (see e.g. [66, 65]).

A promising field is BSP algorithm engineering, as BSP algorithm implementations have great potential to provide equal or better performance compared to standard approaches. Another advantage of BSP is the predictability of the running time. This predictability can be used to determine thresholds for using different algorithm implementations which might have better performance for different input sizes, depending on the relative efficiency of the individual processors and the communication network. Therefore, the next step in enabling the scientific computing community to use BSP is to provide efficient BSP-based algorithmic components. There are different projects that implement parallel graph algorithms [13, 33] or algorithms in numerical linear algebra [42] using BSP-style communication. Other areas of interest are computational biology, parallel string processing, and parallel combinatorial optimization.

Overall, the BSP model has established itself as an accurate theoretical model for parallel computation, as well as a good basis for practical parallel programming. However, more work has to be done in making BSP-based tools and algorithms available for scientific computing, and also in adapting these to changing paradigms in parallel computing. The scientific computing community in particular can benefit from these developments, since optimized

BSP algorithms can already achieve speedup for small problem sizes. Making standard algorithm and communication libraries efficient and portable to modern parallel systems significantly contributes to a wider acceptance of BSP.

Appendix A

The BSPWrapper Framework

A.1 Extended Communication Library Interface

Files:

include/BSPWrapper.h,

BSPWrapper/generic/*,

include/memory.h,

include/paramfile.h,

include/BSPUtils.h,

include/avector.h, include/error.h,

include/multiqueue.h

Unbuffered BSMP

In the MPI implementation, BSPlib-standard functions have been supplemented by the following unbuffered messaging functions that help to conserve memory. Their basic behavior is the same as of the Send and Move functions, but they do not buffer any data. Hence, the data passed to HpSend must not be changed until the end of the superstep. Data pointers obtained by HpMove are only valid in the superstep in which HpMove is called.

```
/* BSMP primitives */
void HpSend(int pid, void* tag, void* message, int count);
```

```
void HpMove(void *& data, int & size);
```

Other Functions

```
/* Helpers */
double Time();
void Print(int pid, const char * format, ...);
```

`Time()` returns the time that has passed since starting the program in seconds. `Print()` is a wrapper for doing a `printf` only on node `pid`. If `pid` < 0, the `printf` command is executed on every node.

Utility Classes

class CAVector $<$ _t $>$

The BSPWrapper utility library further contains the class `CAVector`, a basic C++ templated vector class with a public data pointer that can be passed to BSPWrapper's communication primitives.

Construction and destruction:

The vector can be created empty, from an existing buffer (which will not be deleted upon destruction) or with a specified size. The variable `grow`/_g specifies the alignment when the buffer grows. The size of the vector will always be a multiple of this value.

```
CAVector();
CAVector(int cnt, _t* buffer);
CAVector(int _size, int _g= 1);

~CAVector();
```

Element access:

The individual data elements are accessible via the overloaded `operator[]`. The functions `Put` and `Get` allow safe access to a sequence of elements. If `Put` has to write an array of

elements which exceeds the length of the vector, then the vector will be resized. Get returns the number of elements which were successfully read.

```
_t& operator[](int index);

void Put(const _t* _data, int index, int len) ;
int Get(int index, _t* _data, int len) ;
```

Modifying the size:

SetSize is used to change the size and alignment of the vector. Shrink reduces the vector's length to newcount if it is longer. Both functions preserve the contents of the vector up to the new length.

```
void SetSize(int _size, int _grow= 1);
void Shrink(int newcount);
```

class CMultiQueue < _t >

Based on CAVector, a simple message buffer class, CMultiQueue was created. It manages a set of queues, which can be retrieved as an array and a set of counts and displacements suitable for being transferred by a call to MPI_Alltoallv.

Construction and destruction:

A CMultiQueue can be constructed as an empty message buffer (qcount specifies the number of destinations, qbmin and qbgrow are the initial sizes and alignments of each buffer.

Alternatively, it can be constructed from an array and counts/displacements as obtained after a call to MPI_Alltoallv. None of the buffers which are passed to it will be deleted on destruction.

```
CMultiQueue (int qcount= 0, int _qbmin= 0, int _qbgrow= 1);
CMultiQueue (int qcount, void* array, int counts[], int displs[]);
~CMultiQueue ();
```

Changing the size:

As above, qcount specifies the number of destinations, qbmin and qbgrow are the initial sizes and alignments of each buffer. Reset empties queues and resets them to their initial size.

```
void SetSize(int qcount, int _qbmin= 0, int _qbgrow= 1);
void Reset();
```

Queue access:

Enqueue is used to add an array of data to a specific queue. Head retrieves a pointer to the first element and the remaining length of a specific queue. DeleteHead removes a number of elements from a specific queue. FirstHead and DeleteFirstHead work in a similar way on the first queue that contains data.

```
void Enqueue(int qnr, _t* data, int count) ;
void Head(int qnr, _t*& data, int &count) ;
void DeleteHead(int qnr, int _count) ;
void FirstHead(_t*& data, int &count) ;
void DeleteFirstHead(int count) ;
```

Array conversion:

The functions ToArray and FromArray are used to convert the queue from or into an array of chars and a set of (char) displacements and counts.

```
void ToArray(char*& array, int *& charcounts, int *& chardispls) ;
void FromArray(int qcount, void* array, int counts[], int displs[]) ;
```

A.2 Benchmarking Library

Files: include/BSPBenchmark.h, BSPWrapper/bench/*

A set of benchmarking functions are contained in the namespace BSPBenchmark.

Measuring f

The speed of a single processor can be measured with the following four functions, which return the *time per flop* f in seconds as a double precision float. This time can be estimated for a dot product or a dense matrix-matrix multiplication, both for integers and for double precision floats. If `bench_f.cpp` is compiled with `-D_USE_BLAS`, the BLAS primitives `ddot`/`dgemm` will be used by `measure_*_double`. The matrix multiplication and dot product code was adapted from the original `bspprobe` from the Oxford BSP Toolset [76].

```
/*
 * functions for measuring the inverse flop rate (f)
 * for different types of local computation.
 *
 * the value of f will be returned in seconds
 *
 * */
double measure_f_dot_int();
double measure_f_matmul_int();
double measure_f_dot_double();
double measure_f_matmul_double();
```

Measuring g

The bandwidth can be sampled for each communication primitive and for two different kinds of communication. For the random permutation measurements to work correctly, each processor must have the same starting random seed using `srand` before the call.

The functions return an estimation for g, which is the average for the samples where $g < 2g_{min}$. A more precise estimation can be obtained when using the output data (a file named `bspprobe_<arch>_p<P>_<fn>_[random|alltoall].dat` is created by the function) and the **BSPParam** Perl module (see Appendix B.1).

```
/*
 * functions for measuring the bandwidth (g)
 * for different types of communication.
 *
 * */
double measure_g_alltoall(int size, int h, int fn, double fac,
                          int sub, bool diag= false);
double measure_g_random(int size, int h, int fn, double fac,
                        int sub, bool diag= false);
```

```
/* Functions and names for measure_g_x */
#define HR_FUNCTIONS    5
#define HR_FN_GET   0
#define HR_FN_SEND  1
#define HR_FN_PUT   2
#define HR_FN_HPPUT 3
#define HR_FN_HPGET 4
extern const char * fnames[];
```

Measuring l

The latency can be measured with three kinds of communication preceeding the synchronization:

- No communication. A plain call to the communication library `Sync` function.
- Cyclic shift communication. A cyclic shift (each processor p sends data to processor $(p+1)$ mod P) will be carried out before the synchronization, using `HpPut`.
- All-to-all communication. Each processor p sends data to every other processor using `HpPut` before the synchronization.

All the functions use oversampling, `measure_l_nocomm`, `measure_l_localshift` have been adapted from the original `bspprobe` from the Oxford BSP Toolset [76].

```
/*
 * functions for measuring the latency (l)
 * for different types of communication.
 *
 * the value of l will be returned in seconds
 *
 * */
double measure_l_nocomm();
double measure_l_localshift();
double measure_l_alltoall();
```

Appendix B

BSPWrapper Tools Documentation

B.1 BSPParam - Bandwidth Modeling

The perl module **BSPParam** contains functionality to extract the BSP/LogP bandwidth modelling parameters from a data file that contains raw bandwidth values as measured by BSPWrapper's bspprobe.

The input file has a column format like

```
512     512     0.001
512     256     0.0005
512     128     0.00025
512     64      0.000125
...
```

The first column contains the message size h^*, the second column the number of messages with this size $c = h/h^*$ that were transmitted. The third column must contain the bandwidth gap g or inverse bandwidth measured for this size (i.e. $g = T_{sampled}/(h * \cdot c)$). The following parameters are computed:

Name in Perl script	Variable	Meaning
g_inf	g_∞	Asymptotic value of g
h_half	h_{half}	Estimated value of h_{half}

overhead	o	Per-message overhead
error_bsp	ε_{BSP}	Squared error using enhanced BSP model with h_{half}
error_bspstar	ε_{BSP*}	Squared error using enhanced BSP model with h_{half} and o

B.2 Parsing BSPWrapper/bspprobe Results

parse_bspprobe_data.pl

The perl script will

- Create a performance report using LaTeX.
- Extract BSP and bandwidth modeling parameters for every communication primitive on every system
- Create performance comparison and individual communication profile plots for each communication primitive using gnuplot

Command line parameters

-**cplots** Enable comparison plots (off by default)

-**pplots** Enable individual primitive plots (off by default)

Input data

The script will read its input data from the current directory. Subdirectories must contain the bspprobe output (possibly from different machines). A file **template.gp** must exist, this will be used for setting up gnuplot [78]. Another file **benchex.tex** must exist, it will be used as a wrapper for the auto-generated performance report. Samples are included in the BSPWrapper/Tools distribution.

Appendix C

Communication Benchmark Results

C.1 Results on Shared Memory (`skua`)

Table C.1: Computation speed on `skua`

	Time per Op f	**'Flops'** $1/f$
Value of f for *OXTOOL*		
Dot product (`double`)	0.0034 μs	289.9 M
Matrix product (`double`)	0.0002 μs	5347.6 M
Average (`double`)	0.0004 μs	2818.7 M
Dot product (`int`)	0.0021 μs	478.5 M
Matrix product (`int`)	0.0009 μs	1081.1 M
Average (`int`)	0.0013 μs	779.8 M
Value of f for *MPISHMEM*		
Dot product (`double`)	0.0034 μs	289.9 M
Matrix product (`double`)	0.0002 μs	5347.6 M
Average (`double`)	0.0004 μs	2818.7 M
Dot product (`int`)	0.0020 μs	502.5 M
Matrix product (`int`)	0.0009 μs	1082.3 M
Average (`int`)	0.0013 μs	792.4 M
Value of f for *PUB*		
Dot product (`double`)	0.0036 μs	279.3 M
Matrix product (`double`)	0.0002 μs	5405.4 M
Average (`double`)	0.0004 μs	2842.4 M
Dot product (`int`)	0.0057 μs	176.1 M
Matrix product (`int`)	0.0030 μs	337.8 M
Average (`int`)	0.0039 μs	256.9 M

Table C.2: Latency on skua

p	**no communication**	**cyclic shift**	**all-to-all**
Value of l for *OXTOOL*			
4	11.20 μs	16.90 μs	25.20 μs
8	27.50 μs	33.90 μs	64.10 μs
10	35.60 μs	41.40 μs	81.10 μs
16	62.20 μs	69.70 μs	145.50 μs
25	105.50 μs	110.30 μs	248.50 μs
32	128.80 μs	136.30 μs	327.00 μs
Value of l for *MPISHMEM*			
4	34.40 μs	49.30 μs	44.40 μs
8	43.30 μs	70.50 μs	99.30 μs
10	66.00 μs	94.20 μs	133.60 μs
16	88.50 μs	119.30 μs	207.70 μs
25	163.00 μs	206.80 μs	363.60 μs
32	246.60 μs	273.00 μs	584.20 μs
Value of l for *PUB*			
4	32.90 μs	48.52 μs	82.80 μs
10	61.20 μs	81.80 μs	244.88 μs
16	80.20 μs	100.61 μs	482.30 μs
25	99.90 μs	127.20 μs	984.31 μs
32	97.61 μs	141.38 μs	1459.41 μs

Table C.3: Bandwidth gap on skua

p	**Fn**	g_∞	h_{half}	o	ε_{BSP*}	ε_{BSP}
Random Permutation *Using OXTOOL*						
4	Get	0.04173 μs	354	185	6.4e-10	8.3e-10
4	HpGet	0.02234 μs	594	358	7.7e-10	7.9e-10
4	HpPut	0.02481 μs	544	3	2.2e-11	2.4e-11
4	Put	0.02666 μs	497	8	2.7e-11	3.5e-11
4	Send	0.04418 μs	320	69	1.3e-11	4.2e-10
8	Get	0.04942 μs	615	199	8.1e-10	1.5e-09
8	HpGet	0.02621 μs	1155	365	8.1e-10	1.4e-09
8	HpPut	0.03162 μs	969	4	1.1e-10	1.2e-10

Continued on the next page...

...continued from last page.

p	**Fn**	g_∞	h_{half}	o	ε_{BSP*}	ε_{BSP}
8	Put	0.03477 μs	849	7	1.1e-10	1.2e-10
8	Send	0.06524 μs	468	63	3.0e-11	8.3e-10
10	Get	0.04320 μs	891	228	7.9e-10	1.5e-09
10	HpGet	0.02675 μs	1430	361	7.9e-10	1.4e-09
10	HpPut	0.03235 μs	1113	4	1.8e-10	1.9e-10
10	Put	0.03551 μs	1006	7	1.6e-10	1.8e-10
10	Send	0.05432 μs	687	77	5.2e-11	9.3e-10
16	Get	0.04251 μs	1485	243	7.1e-10	2.1e-09
16	HpGet	0.02746 μs	2293	366	7.1e-10	2.0e-09
16	HpPut	0.03346 μs	1890	131	1.9e-10	1.3e-09
16	Put	0.03665 μs	1687	122	2.2e-10	1.4e-09
16	Send	0.05773 μs	1091	80	2.0e-10	1.4e-09
25	Get	0.04540 μs	2266	233	8.4e-10	2.7e-09
25	HpGet	0.02769 μs	3576	373	1.6e-09	3.5e-09
25	HpPut	0.03419 μs	2858	137	6.2e-10	2.1e-09
25	Put	0.03766 μs	2639	126	5.3e-10	2.0e-09
25	Send	0.09835 μs	1016	50	5.4e-10	2.1e-09
32	Get	0.05358 μs	2515	205	1.1e-09	4.0e-09
32	HpGet	0.02993 μs	4489	363	1.1e-09	4.0e-09
32	HpPut	0.03729 μs	3587	142	1.2e-09	3.3e-09
32	Put	0.03892 μs	3290	131	7.2e-09	9.4e-09
32	Send	0.09180 μs	1409	58	1.1e-09	3.3e-09
Using MPISHMEM						
4	Get	0.01483 μs	2185	28	9.4e-11	1.2e-10
4	HpGet	0.01199 μs	2675	14	9.8e-11	1.1e-10
4	HpPut	0.01090 μs	2958	16	1.0e-10	1.1e-10
4	Put	0.01499 μs	2196	52	8.7e-11	1.5e-10
4	Send	0.06365 μs	500	298	7.7e-09	3.1e-09
8	Get	0.02198 μs	2249	21	2.5e-10	3.0e-10
8	HpGet	0.01479 μs	3367	14	1.9e-10	2.1e-10
8	HpPut	0.01348 μs	3500	18	2.7e-10	2.9e-10
8	Put	0.02118 μs	2502	46	1.8e-10	2.9e-10
8	Send	0.62080 μs	119	7	4.0e-10	8.8e-10
10	Get	0.02423 μs	2880	21	4.5e-10	5.2e-10
10	HpGet	0.01508 μs	4543	17	4.8e-10	5.1e-10
10	HpPut	0.01420 μs	4861	21	4.7e-10	5.1e-10
10	Put	0.02261 μs	3156	45	4.2e-10	5.7e-10
10	Send	0.79223 μs	132	6	7.0e-10	1.3e-09
16	Get	0.01893 μs	4973	30	8.8e-10	9.9e-10

Continued on the next page...

...continued from last page.

p	**Fn**	g_∞	h_{half}	o	ε_{BSP*}	ε_{BSP}
16	HpGet	0.01729 μs	5456	18	9.2e-10	9.8e-10
16	HpPut	0.01516 μs	6474	23	8.3e-10	8.9e-10
16	Put	0.02366 μs	4068	46	8.7e-10	1.1e-09
16	Send	1.20087 μs	128	4	1.6e-09	2.4e-09
25	Get	0.02227 μs	7164	31	2.7e-09	2.9e-09
25	HpGet	0.01633 μs	9792	27	2.7e-09	2.9e-09
25	HpPut	0.01521 μs	11034	33	2.2e-09	2.3e-09
25	Put	0.02343 μs	7031	52	2.4e-09	2.8e-09
25	Send	2.08949 μs	128	3	5.6e-09	7.6e-09
32	Get	0.02316 μs	10055	36	5.3e-09	5.6e-09
32	HpGet	0.01709 μs	13588	35	5.6e-09	5.9e-09
32	HpPut	0.01821 μs	12904	35	5.4e-09	5.7e-09
32	Put	0.02433 μs	9762	57	5.0e-09	5.6e-09
32	Send	0.07003 μs	3304	322	9.5e-09	9.4e-09
Using PUB						
4	Get	0.06713 μs	856	480	2.8e-08	7.8e-09
4	HpGet	0.06669 μs	912	1773	3.8e-07	1.1e-07
4	HpPut	0.06585 μs	855	18	2.3e-10	3.7e-10
4	Put	0.07428 μs	756	15	2.4e-10	3.6e-10
4	Send	0.07395 μs	774	23	2.1e-10	4.2e-10
10	Get	0.05103 μs	1532	129	2.0e-10	1.7e-09
10	HpGet	0.04533 μs	1919	401	3.0e-09	5.2e-09
10	HpPut	0.03677 μs	2057	35	4.3e-10	6.4e-10
10	Put	0.04421 μs	1695	27	4.7e-10	6.7e-10
10	Send	0.04585 μs	1671	37	4.2e-10	7.3e-10
16	Get	0.04786 μs	2517	141	6.7e-10	2.8e-09
16	HpGet	0.04632 μs	2705	423	3.6e-09	6.4e-09
16	HpPut	0.04569 μs	2665	32	1.2e-09	1.5e-09
16	Put	0.05414 μs	2256	24	1.2e-09	1.5e-09
16	Send	0.05272 μs	2294	38	1.2e-09	1.6e-09
25	Get	0.04734 μs	11942	155	2.3e-07	2.2e-07
25	Put	0.05289 μs	2883	26	2.1e-09	2.5e-09
25	Send	0.05213 μs	2949	40	1.9e-09	2.6e-09
32	Get	0.04551 μs	3569	161	1.4e-09	4.1e-09
32	HpGet	0.04274 μs	4065	538	6.2e-09	8.8e-09
32	HpPut	0.04223 μs	3829	38	2.3e-09	2.8e-09
32	Put	0.04866 μs	3324	29	2.4e-09	2.8e-09
32	Send	0.04833 μs	3367	44	2.2e-09	2.9e-09
All to all						

Continued on the next page...

...continued from last page.

p	**Fn**	g_∞	h_{half}	o	ε_{BSP*}	ε_{BSP}
Using OXTOOL						
4	Get	0.04521 μs	177	130	1.3e-10	7.0e-10
4	HpGet	0.02613 μs	292	208	1.1e-10	5.9e-10
4	HpPut	0.02472 μs	284	4	1.1e-11	1.3e-11
4	Put	0.03742 μs	199	4	9.2e-12	1.3e-11
4	Send	0.05137 μs	150	72	1.5e-11	4.8e-10
8	Get	0.04922 μs	173	114	7.1e-11	7.3e-10
8	HpGet	0.02566 μs	313	209	6.4e-11	6.6e-10
8	HpPut	0.02801 μs	276	5	1.3e-11	1.6e-11
8	Put	0.03747 μs	207	4	1.3e-11	1.6e-11
8	Send	0.04912 μs	169	83	2.3e-11	5.7e-10
10	Get	0.03963 μs	214	137	5.2e-11	6.8e-10
10	HpGet	0.02331 μs	346	220	4.7e-11	6.1e-10
10	HpPut	0.02725 μs	287	5	1.2e-11	1.6e-11
10	Put	0.03585 μs	215	4	1.3e-11	1.7e-11
10	Send	0.04935 μs	173	87	2.3e-11	5.5e-10
16	Get	0.04015 μs	225	144	5.6e-11	7.9e-10
16	HpGet	0.02610 μs	344	210	5.2e-11	7.1e-10
16	HpPut	0.02960 μs	297	150	2.3e-11	6.0e-10
16	Put	0.03747 μs	234	121	2.4e-11	6.2e-10
16	Send	0.05106 μs	174	93	2.8e-11	6.7e-10
25	Get	0.27693 μs	34	21	6.4e-10	1.4e-09
25	HpGet	0.02856 μs	322	244	2.8e-10	1.1e-09
25	HpPut	0.04291 μs	213	114	6.0e-10	1.4e-09
25	Put	0.04108 μs	227	124	3.1e-11	7.2e-10
25	Send	0.05204 μs	185	103	3.7e-11	7.8e-10
32	Get	0.07925 μs	129	81	6.3e-11	8.9e-10
32	HpGet	0.03263 μs	312	187	5.1e-11	8.5e-10
32	HpPut	0.03973 μs	249	135	3.4e-11	7.6e-10
32	Put	0.04770 μs	333	109	4.7e-09	5.8e-09
32	Send	0.06374 μs	158	87	4.9e-11	8.2e-10
Using MPISHMEM						
4	Get	0.00350 μs	3336	154	5.5e-12	2.1e-11
4	HpGet	0.00274 μs	3869	67	9.6e-12	1.4e-11
4	HpPut	0.01199 μs	918	15	1.3e-11	1.8e-11
4	Put	0.01989 μs	598	46	7.8e-12	5.0e-11
4	Send	0.12475 μs	98	55	5.2e-10	6.6e-10
8	Get	0.00258 μs	2740	188	3.0e-12	1.5e-11
8	HpGet	0.00161 μs	4371	116	3.5e-12	6.2e-12

Continued on the next page...

...continued from last page.

p	**Fn**	g_∞	h_{half}	o	ε_{BSP*}	ε_{BSP}
8	HpPut	0.01256 μs	618	15	7.0e-12	1.1e-11
8	Put	0.02035 μs	420	44	2.8e-12	3.7e-11
8	Send	0.21597 μs	68	38	8.3e-10	6.6e-10
10	Get	0.00246 μs	3310	202	2.7e-12	1.3e-11
10	HpGet	0.00153 μs	5170	140	3.8e-12	6.9e-12
10	HpPut	0.01375 μs	628	15	8.0e-12	1.2e-11
10	Put	0.02190 μs	427	42	3.8e-12	3.8e-11
10	Send	0.22082 μs	69	34	6.3e-10	5.5e-10
16	Get	0.00193 μs	3483	247	1.6e-12	1.1e-11
16	HpGet	0.00131 μs	4901	147	2.9e-12	5.5e-12
16	HpPut	0.01383 μs	535	14	9.7e-12	1.4e-11
16	Put	0.02158 μs	387	43	5.4e-12	4.1e-11
16	Send	0.28425 μs	46	42	2.1e-09	1.2e-09
25	Get	0.00192 μs	3644	251	2.2e-12	1.2e-11
25	HpGet	0.00122 μs	5700	174	3.2e-12	5.8e-12
25	HpPut	0.01405 μs	538	14	7.0e-12	1.1e-11
25	Put	0.02196 μs	263	43	3.1e-11	8.3e-11
25	Send	0.34002 μs	48	48	4.0e-09	2.1e-09
32	Get	0.00164 μs	4733	303	2.5e-12	1.2e-11
32	HpGet	0.00130 μs	5954	176	3.8e-12	6.9e-12
32	HpPut	0.01592 μs	539	14	7.7e-12	1.2e-11
32	Put	0.02287 μs	402	42	3.7e-12	3.8e-11
32	Send	0.83983 μs	10	3	3.7e-09	3.7e-09
Using PUB						
4	Get	0.15332 μs	137	623	2.3e-07	6.5e-08
4	HpGet	0.14587 μs	373	1683	8.8e-07	6.8e-07
4	HpPut	0.15182 μs	132	10	5.7e-11	1.5e-10
4	Put	0.15921 μs	128	9	3.3e-11	1.1e-10
4	Send	0.15897 μs	136	14	3.2e-11	1.7e-10
10	Get	0.08557 μs	172	252	5.8e-09	4.0e-09
10	HpGet	0.14331 μs	164	528	8.8e-08	4.5e-08
10	HpPut	0.04409 μs	242	30	2.9e-12	6.7e-11
10	Put	0.04464 μs	238	28	3.1e-12	6.0e-11
10	Send	0.04656 μs	258	38	8.7e-13	9.6e-11
16	Get	0.17902 μs	85	191	1.8e-08	9.6e-09
16	HpGet	0.17537 μs	119	828	3.9e-07	1.5e-07
16	HpPut	0.13499 μs	84	10	6.2e-12	6.2e-11
16	Put	0.13069 μs	85	10	7.5e-12	5.8e-11
16	Send	0.14456 μs	86	14	8.6e-12	1.0e-10

Continued on the next page...

...continued from last page.

p	**Fn**	g_∞	h_{half}	o	ε_{BSP*}	ε_{BSP}
25	Get	0.24453 μs	56	178	3.1e-08	1.4e-08
25	Put	0.19776 μs	49	6	1.3e-11	4.3e-11
25	Send	0.20955 μs	50	9	1.8e-11	8.5e-11
32	Get	0.30795 μs	40	159	3.7e-08	1.8e-08
32	HpGet	0.30443 μs	64	647	6.5e-07	2.8e-07
32	HpPut	0.23150 μs	37	5	1.5e-11	4.7e-11
32	Put	0.23288 μs	37	5	1.7e-11	4.5e-11
32	Send	0.23564 μs	42	8	2.0e-11	7.9e-11

C.2 Results on Distributed Memory, Ethernet (`argus`)

Table C.4: Computation speed on `argus`

	Time per Op f	**'Flops'** $1/f$
Value of f for *OXTOOL*		
Dot product (`double`)	0.0104 μs	96.2 M
Matrix product (`double`)	0.0027 μs	371.7 M
Average (`double`)	0.0043 μs	234.0 M
Dot product (`int`)	0.0056 μs	179.5 M
Matrix product (`int`)	0.0012 μs	806.5 M
Average (`int`)	0.0020 μs	493.0 M
Value of f for *PUB*		
Dot product (`double`)	0.0094 μs	106.4 M
Matrix product (`double`)	0.0004 μs	2298.9 M
Average (`double`)	0.0008 μs	1202.6 M
Dot product (`int`)	0.0055 μs	181.5 M
Matrix product (`int`)	0.0012 μs	813.0 M
Average (`int`)	0.0020 μs	497.2 M
Value of f for *MPIMPASS*		
Dot product (`double`)	0.1880 μs	5.3 M
Matrix product (`double`)	0.0005 μs	2049.2 M
Average (`double`)	0.0010 μs	1027.2 M
Dot product (`int`)	0.0057 μs	175.4 M
Matrix product (`int`)	0.0012 μs	806.5 M
Average (`int`)	0.0020 μs	490.9 M

Table C.5: Latency on `argus`

p	**no communication**	**cyclic shift**	**all-to-all**
Value of l for *OXTOOL*			
4	808.00 μs	1599.20 μs	1568.90 μs
10	614.00 μs	1047.40 μs	1565.90 μs
Value of l for *PUB*			
4	983.00 μs	4033.70 μs	1372.80 μs
10	2027.40 μs	5600.20 μs	3035.20 μs
Value of l for *MPIMPASS*			
4	7854.40 μs	7294.20 μs	8100.20 μs
10	18754.60 μs	18197.30 μs	19270.00 μs

Table C.6: Bandwidth gap on argus

p	**Fn**	g_∞	h_{half}	o	ε_{BSP*}	ε_{BSP}
Random Permutation						
Using OXTOOL						
4	Get	1.78528 μs	576	320	5.6e-06	1.7e-05
4	HpGet	3.63672 μs	255	214	7.5e-06	2.2e-05
4	HpPut	3.48879 μs	288	1	1.4e-07	1.5e-07
4	Put	3.58051 μs	280	2	1.8e-07	1.9e-07
4	Send	3.17405 μs	266	173	6.2e-06	1.9e-05
10	Get	6.32309 μs	205	240	2.2e-04	4.2e-04
10	HpGet	6.36348 μs	207	223	1.7e-04	3.5e-04
10	HpPut	6.09979 μs	166	1	1.8e-07	1.9e-07
10	Put	5.35441 μs	157	1	1.8e-07	1.9e-07
10	Send	6.79959 μs	196	183	1.9e-04	3.5e-04
Using PUB						
4	Get	2.44785 μs	443	286	5.7e-06	1.5e-05
4	HpGet	2.96265 μs	343	444	1.3e-05	6.2e-05
4	Put	1.21292 μs	850	20	7.3e-08	1.1e-07
4	Send	2.63778 μs	325	23	7.3e-08	2.8e-07
10	Get	5.78395 μs	469	202	7.9e-05	1.9e-04
10	HpGet	6.33894 μs	961	323	2.4e-04	5.7e-04
10	HpPut	6.30675 μs	588	5	1.9e-06	1.8e-06
10	Put	5.53043 μs	607	7	2.3e-06	2.3e-06
10	Send	6.02710 μs	397	16	9.1e-07	1.2e-06
Using MPIMPASS						
4	Get	2.74347 μs	1950	249	4.3e-06	2.3e-05
4	HpGet	3.24968 μs	1831	218	4.0e-06	1.9e-05
4	HpPut	4.18889 μs	1398	171	4.4e-06	2.0e-05
4	Put	1.27564 μs	4336	462	3.1e-06	2.1e-05
4	Send	3.57495 μs	1613	9	3.0e-06	3.3e-06
10	Get	6.30124 μs	2531	286	3.3e-05	1.6e-04
10	HpGet	7.11564 μs	2280	235	5.0e-05	1.7e-04
10	HpPut	7.92463 μs	4101	213	4.2e-04	3.4e-04
10	Put	9.15032 μs	1743	178	8.8e-06	1.0e-04
10	Send	7.99085 μs	1982	11	2.5e-05	2.7e-05
All to all						
Using OXTOOL						
4	Get	1.53880 μs	272	387	3.4e-06	5.9e-06
4	HpGet	4.39223 μs	90	200	9.5e-06	9.7e-06

Continued on the next page...

...continued from last page.

p	**Fn**	g_∞	h_{half}	o	ε_{BSP*}	ε_{BSP}
4	HpPut	4.83696 μs	90	1	1.9e-08	2.0e-08
4	Put	4.56499 μs	93	1	2.2e-08	2.4e-08
10	Get	2.10858 μs	70	514	1.5e-05	1.2e-05
10	HpGet	6.09863 μs	22	179	1.5e-05	1.2e-05
10	HpPut	5.98969 μs	22	-0	2.0e-08	2.0e-08
10	Put	6.61703 μs	18	-0	7.2e-09	7.2e-09
10	Send	5.98250 μs	25	179	1.4e-05	1.1e-05
Using PUB						
4	HpGet	5.25472 μs	68	203	8.2e-06	4.4e-05
4	HpPut	3.73596 μs	84	6	1.6e-08	2.7e-08
4	Put	3.47809 μs	79	7	1.5e-08	2.9e-08
4	Send	3.56861 μs	79	14	1.0e-08	7.0e-08
10	Get	3.25443 μs	75	226	6.1e-06	5.2e-06
10	HpGet	7.14679 μs	32	230	2.3e-05	3.4e-05
10	HpPut	4.28575 μs	68	4	5.9e-09	1.0e-08
10	Put	4.17672 μs	66	4	5.5e-09	1.1e-08
10	Send	4.12093 μs	72	11	1.1e-08	3.3e-08
Using MPIMPASS						
4	Get	1.66011 μs	1305	460	8.3e-06	1.3e-05
4	HpGet	5.75424 μs	331	131	9.6e-06	1.8e-05
4	HpPut	3.91307 μs	490	75	6.8e-07	1.8e-06
4	Put	3.65087 μs	522	121	2.1e-06	3.0e-06
4	Send	4.33270 μs	457	5	3.4e-07	4.0e-07
10	Get	3.33746 μs	552	310	1.2e-05	1.0e-05
10	HpGet	10.25375 μs	171	99	9.2e-06	1.1e-05
10	HpPut	4.91546 μs	352	46	2.3e-07	8.9e-07
10	Put	5.21218 μs	329	51	3.9e-07	1.1e-06
10	Send	4.88309 μs	375	7	2.4e-07	3.1e-07

C.3 Results on Distributed Memory, Myrinet (`aracari`)

Table C.7: Computation speed on `aracari`

	Time per Op f	**'Flops'** $1/f$
Value of f for *OXTOOL*		
Dot product (`double`)	0.0245 μs	40.8 M
Matrix product (`double`)	0.0016 μs	625.0 M
Average (`double`)	0.0030 μs	332.9 M
Dot product (`int`)	0.0118 μs	84.7 M
Matrix product (`int`)	0.0017 μs	591.7 M
Average (`int`)	0.0030 μs	338.2 M
Value of f for *PUB*		
Dot product (`double`)	0.0251 μs	39.8 M
Matrix product (`double`)	0.0024 μs	413.2 M
Average (`double`)	0.0044 μs	226.5 M
Dot product (`int`)	0.0119 μs	84.0 M
Matrix product (`int`)	0.0022 μs	450.5 M
Average (`int`)	0.0037 μs	267.2 M
Value of f for *MPIMPASS*		
Dot product (`double`)	0.0300 μs	33.3 M
Matrix product (`double`)	0.0017 μs	598.8 M
Average (`double`)	0.0032 μs	316.1 M
Dot product (`int`)	0.0121 μs	82.6 M
Matrix product (`int`)	0.0017 μs	602.4 M
Average (`int`)	0.0029 μs	342.5 M

Table C.8: Latency on `aracari`

p	**no communication**	**cyclic shift**	**all-to-all**
Value of l for *OXTOOL*			
4	43.40 μs	64.50 μs	87.50 μs
8	122.00 μs	146.80 μs	249.70 μs
10	164.20 μs	188.30 μs	331.70 μs
16	283.90 μs	309.50 μs	575.00 μs
32	607.70 μs	634.40 μs	1227.00 μs
Value of l for *PUB*			
4	56.79 μs	75.60 μs	100.40 μs
8	102.62 μs	122.79 μs	237.08 μs

Continued on the next page...

...continued from last page.

p	**no communication**	**cyclic shift**	**all-to-all**
10	102.31 μs	122.79 μs	278.90 μs
16	172.90 μs	189.71 μs	482.80 μs
32	360.49 μs	375.99 μs	986.91 μs
Value of l for *MPIMPASS*			
4	333.10 μs	343.90 μs	384.80 μs
8	731.60 μs	798.10 μs	956.80 μs
10	1102.40 μs	1129.90 μs	1323.30 μs
16	1447.30 μs	1482.40 μs	1819.80 μs
32	2778.40 μs	2805.00 μs	3578.10 μs

Table C.9: Bandwidth gap on `aracari`

p	**Fn**	g_∞	h_{half}	o	ε_{BSP*}	ε_{BSP}
Random Permutation						
Using OXTOOL						
4	Get	0.18552 μs	249	142	9.2e-09	8.5e-09
4	HpGet	0.12560 μs	353	217	1.1e-08	8.2e-09
4	HpPut	0.16114 μs	273	1	2.0e-10	2.1e-10
4	Put	0.14070 μs	327	3	2.2e-10	2.5e-10
4	Send	0.23052 μs	196	37	1.3e-10	2.9e-09
8	Get	0.25679 μs	477	153	1.1e-08	2.6e-08
8	HpGet	0.21733 μs	574	181	1.2e-08	2.5e-08
8	HpPut	0.24953 μs	488	2	1.4e-09	1.5e-09
8	Put	0.25384 μs	475	2	1.6e-09	1.7e-09
8	Send	0.30818 μs	404	61	1.6e-10	1.5e-08
10	Get	0.26088 μs	604	160	1.2e-08	3.0e-08
10	HpGet	0.24548 μs	655	170	1.2e-08	3.1e-08
10	HpPut	1.28570 μs	121	-0	7.9e-09	7.9e-09
10	Put	0.26453 μs	597	2	2.3e-09	2.5e-09
10	Send	0.34736 μs	495	60	3.5e-10	1.8e-08
16	Get	0.29210 μs	903	145	9.7e-09	3.7e-08
16	HpGet	0.27320 μs	1268	186	1.1e-05	1.1e-05
16	HpPut	0.83624 μs	314	31	3.4e-07	3.8e-07
16	Put	0.28516 μs	922	84	4.3e-08	7.4e-08
16	Send	0.35500 μs	735	65	3.1e-09	2.9e-08
32	Get	0.33669 μs	1675	137	5.6e-09	2.4e-08
32	HpGet	0.48880 μs	1155	171	5.4e-07	6.8e-07
32	HpPut	0.49472 μs	1141	379	4.0e-06	4.0e-06

Continued on the next page...

...continued from last page.

p	**Fn**	g_∞	h_{half}	o	ε_{BSP*}	ε_{BSP}
32	Put	0.31482 μs	1779	70	6.4e-09	2.0e-08
32	Send	0.38172 μs	1474	59	5.9e-09	1.9e-08
Using PUB						
4	Get	0.78523 μs	90	564	5.2e-06	1.7e-06
4	HpGet	0.29796 μs	255	3458	2.9e-05	8.1e-06
4	HpPut	0.23395 μs	289	640	5.9e-07	2.4e-07
4	Put	0.78058 μs	89	192	6.0e-07	2.4e-07
4	Send	0.77379 μs	99	771	9.5e-06	2.7e-06
8	Get	0.79534 μs	151	555	5.0e-06	1.7e-06
8	HpGet	0.30172 μs	463	3430	2.9e-05	8.2e-06
8	HpPut	0.27693 μs	404	541	5.6e-07	2.5e-07
8	Put	0.82844 μs	136	181	5.6e-07	2.5e-07
8	Send	0.78951 μs	156	765	9.5e-06	2.8e-06
10	Get	0.79287 μs	255	571	5.3e-06	1.8e-06
10	HpGet	0.31990 μs	441	3314	3.0e-05	8.6e-06
10	HpPut	0.29469 μs	410	520	5.8e-07	2.6e-07
10	Put	0.82601 μs	149	185	5.9e-07	2.6e-07
10	Send	0.82042 μs	164	738	9.5e-06	2.8e-06
16	Get	0.79853 μs	226	561	5.1e-06	1.8e-06
16	HpGet	0.32802 μs	596	3222	3.0e-05	8.5e-06
16	HpPut	0.31178 μs	555	490	5.7e-07	2.6e-07
16	Put	0.84594 μs	206	180	5.6e-07	2.5e-07
16	Send	0.83204 μs	223	730	9.5e-06	2.8e-06
32	Get	0.79423 μs	452	638	2.5e-06	6.8e-07
32	HpGet	0.33143 μs	1108	3590	1.5e-05	3.4e-06
32	HpPut	0.32392 μs	1092	531	2.6e-07	9.6e-08
32	Put	0.83117 μs	434	207	2.5e-07	1.1e-07
32	Send	0.84371 μs	444	810	4.7e-06	1.2e-06
Using MPIMPASS						
4	Get	0.20144 μs	1473	208	2.1e-08	3.4e-08
4	HpGet	0.13358 μs	2180	342	2.7e-08	3.7e-08
4	HpPut	0.14582 μs	1879	461	7.8e-08	5.3e-08
4	Put	0.19498 μs	1436	441	1.4e-07	7.8e-08
4	Send	0.24374 μs	1123	83	7.9e-09	1.1e-08
8	Get	0.27458 μs	2460	192	3.4e-08	1.1e-07
8	HpGet	0.25416 μs	2644	223	3.7e-08	1.1e-07
8	HpPut	0.24379 μs	2795	329	8.6e-08	1.1e-07
8	Put	0.32503 μs	1922	319	1.5e-07	1.6e-07
8	Send	0.30741 μs	1993	77	2.6e-08	4.4e-08

Continued on the next page...

...continued from last page.

p	**Fn**	g_∞	h_{half}	o	ε_{BSP*}	ε_{BSP}
10	Get	0.29266 μs	3300	185	6.7e-08	1.8e-07
10	HpGet	0.28895 μs	3347	190	6.1e-08	1.7e-07
10	HpPut	0.25664 μs	3615	300	7.8e-08	1.8e-07
10	Put	0.36752 μs	2564	273	1.3e-07	2.0e-07
10	Send	0.34293 μs	2690	67	6.6e-08	9.8e-08
16	Get	0.29195 μs	4370	190	1.4e-07	3.0e-07
16	HpGet	0.29416 μs	4333	210	1.1e-07	2.6e-07
16	HpPut	0.27372 μs	4468	319	1.2e-07	2.6e-07
16	Put	0.35576 μs	3455	327	1.9e-07	3.3e-07
16	Send	0.34771 μs	3607	79	1.1e-07	1.5e-07
32	Get	0.34198 μs	7327	207	1.6e-07	3.0e-07
32	HpGet	0.33610 μs	7457	216	1.6e-07	3.0e-07
32	HpPut	0.28305 μs	8694	311	1.4e-07	2.8e-07
32	Put	0.39751 μs	6189	304	1.4e-07	2.9e-07
32	Send	0.37784 μs	6442	72	2.0e-07	2.5e-07
All to all						
Using OXTOOL						
4	Get	0.29446 μs	87	87	4.6e-09	1.0e-08
4	HpGet	0.18398 μs	140	134	7.5e-09	1.5e-08
4	HpPut	0.17087 μs	145	1	5.8e-11	6.8e-11
4	Put	0.26765 μs	90	1	6.1e-11	7.0e-11
4	Send	0.35583 μs	73	42	4.0e-10	6.1e-09
8	Get	0.37209 μs	84	70	2.8e-09	1.2e-08
8	HpGet	0.27578 μs	114	91	2.5e-09	1.1e-08
8	HpPut	0.27490 μs	109	1	8.4e-11	9.5e-11
8	Put	0.37763 μs	79	0	9.0e-11	9.7e-11
8	Send	0.44444 μs	70	42	5.9e-10	8.8e-09
10	Get	0.36992 μs	87	67	2.0e-09	1.1e-08
10	HpGet	0.29085 μs	111	82	1.8e-09	1.1e-08
10	HpPut	0.27672 μs	110	1	1.1e-10	1.2e-10
10	Put	0.35963 μs	85	1	9.3e-11	1.0e-10
10	Send	0.46019 μs	70	42	6.2e-10	8.7e-09
16	Get	0.42269 μs	91	71	3.1e-09	1.6e-08
16	HpGet	0.36214 μs	92	71	2.2e-09	1.2e-08
16	HpPut	0.33082 μs	101	68	2.3e-09	1.3e-08
16	Put	0.42921 μs	78	48	8.0e-10	9.6e-09
16	Send	0.52310 μs	64	40	8.0e-10	9.8e-09
32	Get	0.33502 μs	108	80	1.3e-09	4.0e-09
32	HpGet	0.28111 μs	128	91	1.2e-09	3.7e-09

Continued on the next page...

...continued from last page.

p	**Fn**	g_∞	h_{half}	o	ε_{BSP*}	ε_{BSP}
32	HpPut	0.27500 μs	132	79	6.4e-10	3.1e-09
32	Put	0.36643 μs	98	60	6.6e-10	3.2e-09
32	Send	0.48371 μs	75	47	7.6e-10	3.3e-09
Using PUB						
4	Get	1.38399 μs	26	225	2.0e-06	8.6e-07
4	HpGet	0.68405 μs	73	1025	1.1e-05	3.8e-06
4	HpPut	0.60117 μs	53	182	2.5e-07	1.3e-07
4	Put	1.38077 μs	22	79	2.5e-07	1.3e-07
4	Send	1.35679 μs	34	305	3.6e-06	1.4e-06
8	Get	1.33408 μs	31	142	6.1e-07	3.2e-07
8	HpGet	0.74805 μs	81	561	3.2e-06	1.4e-06
8	HpPut	0.68551 μs	61	99	7.0e-08	5.0e-08
8	Put	1.34738 μs	24	50	7.0e-08	5.3e-08
8	Send	1.33974 μs	36	188	1.1e-06	5.2e-07
10	Get	1.18424 μs	35	121	2.9e-07	1.9e-07
10	HpGet	0.63618 μs	94	491	1.6e-06	7.8e-07
10	Put	1.22625 μs	25	43	3.6e-08	3.1e-08
10	Send	1.21961 μs	39	156	5.4e-07	3.0e-07
16	Get	1.33351 μs	33	104	2.7e-07	1.8e-07
16	HpGet	0.81662 μs	75	372	1.5e-06	7.5e-07
16	HpPut	0.77098 μs	39	66	3.0e-08	3.0e-08
16	Put	1.36141 μs	21	37	3.1e-08	2.9e-08
16	Send	1.36567 μs	36	135	5.0e-07	2.8e-07
32	Get	0.82118 μs	56	131	5.2e-08	3.5e-08
32	HpGet	0.40893 μs	157	567	2.8e-07	1.4e-07
32	HpPut	0.43581 μs	68	94	5.9e-09	6.2e-09
32	Put	0.82617 μs	37	49	5.9e-09	6.2e-09
32	Send	0.83142 μs	69	172	9.6e-08	5.8e-08
Using MPIMPASS						
4	Get	0.22792 μs	604	162	7.5e-09	2.8e-08
4	HpGet	0.21700 μs	648	168	6.9e-09	2.7e-08
4	HpPut	0.21873 μs	486	195	1.4e-08	2.7e-08
4	Put	0.40907 μs	257	275	2.1e-07	1.2e-07
4	Send	0.37634 μs	247	27	1.1e-09	2.3e-09
8	Get	0.35052 μs	437	119	4.6e-09	3.8e-08
8	HpGet	0.33369 μs	442	128	5.1e-09	4.0e-08
8	HpPut	0.31761 μs	342	122	5.7e-09	2.8e-08
8	Put	0.46533 μs	235	281	2.5e-07	1.6e-07
8	Send	0.42466 μs	209	18	6.0e-10	1.9e-09

Continued on the next page...

...continued from last page.

p	**Fn**	g_∞	h_{half}	o	ε_{BSP*}	ε_{BSP}
10	Get	0.40692 μs	398	105	4.2e-09	3.9e-08
10	HpGet	0.38795 μs	417	112	4.5e-09	4.0e-08
10	HpPut	0.34484 μs	357	107	3.4e-09	2.6e-08
10	Put	0.47916 μs	267	247	1.8e-07	1.3e-07
10	Send	0.40786 μs	250	17	6.2e-10	1.8e-09
16	Get	0.56758 μs	263	100	1.1e-08	5.9e-08
16	HpGet	0.55811 μs	264	103	1.1e-08	6.2e-08
16	HpPut	0.42995 μs	239	103	8.0e-09	3.2e-08
16	Put	0.59545 μs	223	342	6.6e-07	3.3e-07
16	Send	0.51066 μs	163	14	5.5e-10	1.6e-09
32	Get	0.60649 μs	256	136	1.5e-08	3.1e-08
32	HpGet	0.60302 μs	257	136	1.4e-08	3.2e-08
32	HpPut	0.42796 μs	243	146	1.1e-08	1.5e-08
32	Put	0.59823 μs	174	468	4.7e-07	2.0e-07
32	Send	0.44904 μs	181	16	1.3e-10	4.2e-10

Appendix D

Matrix Multiplication Results — Customized Data Distribution

D.1 Results on Shared Memory (`skua`)

Table D.1: Efficiency and mean relative prediction error on `skua`

	Oxtool			**PUB**			**MPI-2**		
q	Efficiency	rel. error (%)		Efficiency	rel. error (%)		Efficiency	rel. error (%)	
	(%)	$const.\ g$	$g(h,h^*)$	(%)	$const.\ g$	$g(h,h^*)$	(%)	$const.\ g$	$g(h,h^*)$
4 *processors*									
8	78.91	17.4	17.2	70.30	15.3	15.9	78.72	12.6	12.5
64	74.33	12.8	15.5	66.45	35.6	34.6	83.71	22.6	26.3
125	71.07	14.2	13.3	64.35	7.1	7.6	80.97	24.0	23.9
216	72.26	11.0	13.5	65.81	33.7	33.4	80.58	26.2	31.5
343	66.68	8.8	8.8	62.27	35.0	34.2	79.65	27.1	34.3
512	71.11	16.5	22.5	65.62	30.3	30.5	81.99	33.9	39.0
729	63.61	8.3	10.4	58.92	29.8	31.4	76.74	30.0	36.7
10 *processors*									
64	56.34	17.1	12.9	53.20	28.2	25.3	63.42	8.6	9.9
125	61.87	8.6	4.8	57.58	17.0	8.2	71.78	11.9	10.6
216	60.28	18.8	16.1	56.79	29.7	31.0	76.02	17.2	18.8
343	58.12	20.6	18.7	54.53	28.4	35.1	76.73	23.0	23.9
512	55.34	22.2	22.7	52.37	27.3	34.6	73.65	20.7	21.1
729	49.69	26.9	27.9	46.65	29.3	40.3	69.21	16.4	18.2
16 *processors*									

Continued on the next page...

...continued from last page.

	Oxtool			PUB			MPI-2		
q	Efficiency	error (%)		Efficiency	error (%)		Efficiency	error (%)	
	(%)	$const.\ g$	$g(h,h^*)$	(%)	$const.\ g$	$g(h,h^*)$	(%)	$const.\ g$	$g(h,h^*)$
64	64.16	18.5	18.1	57.13	33.0	36.5	78.67	12.8	14.1
125	57.02	10.9	5.9	53.91	21.4	7.8	70.00	18.7	15.9
216	55.74	19.3	17.1	53.01	28.2	31.7	72.78	19.0	19.8
343	51.94	24.4	25.1	48.75	29.8	40.7	69.44	16.3	18.3
512	56.71	14.7	17.3	51.67	26.2	36.0	73.76	24.3	22.1
729	46.31	26.5	30.9	43.36	27.9	44.1	65.99	18.9	18.7
32 *processors*									
125	46.18	57.6	57.8	47.75	53.5	45.9	55.00	48.4	42.4
216	40.50	54.8	56.8	38.59	54.5	66.5	59.49	38.3	33.7
343	47.47	33.6	38.2	39.03	53.6	64.8	50.05	47.3	41.8
512	46.17	46.7	51.4	26.29	52.7	72.8	51.30	47.9	42.1
729	41.01	49.0	54.0	21.50	59.6	74.5	55.73	49.3	40.5

D.2 Results on Distributed Memory, Ethernet (`argus`)

Table D.2: Efficiency and mean relative prediction error on `argus`

	Oxtool			PUB			MPI		
q	Efficiency	rel. error (%)		Efficiency	rel. error (%)		Efficiency	rel. error (%)	
	(%)	$const.\ g$	$g(h,h^*)$	(%)	$const.\ g$	$g(h,h^*)$	(%)	$const.\ g$	$g(h,h^*)$
4 *processors*									
8	36.35	83.9	83.9	34.74	82.7	82.8	-	-	-
64	20.90	84.3	84.3	15.91	79.4	79.6	10.16	64.8	65.7
125	10.67	76.7	76.7	8.09	68.0	68.5	4.71	44.8	46.3
216	14.53	79.2	78.8	11.05	61.4	60.5	7.14	60.9	62.4
343	6.79	66.1	68.9	4.46	43.8	43.2	3.20	38.6	45.6
512	9.83	77.0	79.0	6.65	60.3	62.2	4.60	54.5	59.0
729	5.30	64.8	67.3	3.45	38.0	38.0	2.40	35.7	43.4
10 *processors*									
64	6.71	66.5	66.6	5.98	63.9	61.7	3.58	31.9	32.2
125	5.13	58.6	58.8	5.04	59.7	55.9	2.60	17.9	18.4
216	3.96	51.7	56.1	3.18	34.3	35.5	2.04	10.1	16.7
343	3.59	53.4	58.3	2.91	35.3	37.8	1.80	13.3	20.5
512	2.79	47.1	50.5	2.17	32.4	30.3	1.35	4.1	7.5
729	2.37	41.8	47.8	1.75	25.0	28.2	1.14	7.2	5.7

D.3 Results on Distributed Memory, Myrinet (`aracari`)

Table D.3: Efficiency and mean relative prediction error on `aracari`

	Oxtool			**PUB**			**MPI**		
q	Efficiency	rel. error (%)		Efficiency	rel. error (%)		Efficiency	rel. error (%)	
	(%)	$const.\ g$	$g(h,h^*)$	(%)	$const.\ g$	$g(h,h^*)$	(%)	$const.\ g$	$g(h,h^*)$
4 *processors*									
8	91.20	4.3	4.3	78.08	22.4	22.4	79.20	22.7	22.7
64	76.54	10.2	11.6	60.49	25.6	25.6	66.35	14.3	14.3
125	69.52	9.8	9.7	53.60	23.1	23.0	52.62	20.3	20.4
216	65.56	12.9	16.4	48.78	32.3	29.2	52.39	17.4	17.4
343	62.25	8.5	14.4	45.29	23.8	21.4	46.89	19.8	16.0
512	60.32	18.0	22.2	44.81	25.4	23.3	50.59	11.5	12.9
729	58.23	11.8	18.2	41.42	18.3	17.8	43.11	17.1	13.3
10 *processors*									
64	65.18	14.1	15.5	54.20	14.1	13.2	48.60	16.6	15.1
125	59.71	12.1	10.1	42.09	23.7	22.2	42.54	22.0	23.1
216	56.71	8.9	10.0	43.70	27.3	24.5	39.85	22.4	21.6
343	54.85	15.0	15.9	42.65	19.0	16.0	38.56	17.4	16.5
512	50.83	8.8	8.6	37.09	22.0	18.6	32.44	24.0	23.7
729	46.59	6.7	6.0	35.29	18.7	14.9	30.64	24.7	24.6
16 *processors*									
64	66.39	8.5	8.3	48.16	30.7	29.9	47.89	27.0	27.3
125	59.01	25.4	23.0	46.87	17.9	16.8	39.31	15.9	15.8
216	52.92	18.2	17.9	39.87	28.8	26.4	35.29	19.7	19.5
343	47.98	10.7	10.4	35.95	23.2	19.6	31.99	23.8	23.9
512	42.61	8.3	4.0	26.75	33.6	31.7	26.53	33.1	35.0
729	41.96	9.5	6.7	29.49	19.8	16.0	26.08	26.4	27.8
32 *processors*									
125	50.29	10.2	5.2	40.54	26.3	27.0	36.15	24.5	24.0
216	47.54	12.7	9.1	38.07	61.0	61.7	33.29	30.6	29.3
343	46.14	6.8	2.4	37.84	53.4	54.3	34.88	18.0	15.9
512	39.51	20.2	21.9	30.42	57.3	58.2	28.09	29.4	29.3
729	38.01	12.0	12.8	30.21	56.1	58.0	28.55	21.8	20.9

Appendix E

Matrix Multiplication Results — Static Data Distribution

E.1 Results on Shared Memory (`skua`)

Table E.1: Efficiency and mean relative prediction error on `skua`

	Oxtool			**PUB**			**MPI-2**		
q	Efficiency	rel. error (%)		Efficiency	rel. error (%)		Efficiency	rel. error (%)	
	(%)	$const.\ g$	$g(h,h^*)$	(%)	$const.\ g$	$g(h,h^*)$	(%)	$const.\ g$	$g(h,h^*)$
4 *processors*									
8	75.96	18.3	25.0	67.82	15.6	21.7	76.50	14.7	22.1
64	54.72	38.5	10.4	13.55	87.4	79.3	65.75	8.2	28.5
125	49.84	42.6	5.3	8.23	90.2	83.6	60.92	7.7	36.7
216	49.84	44.7	5.6	9.93	88.4	78.6	67.83	6.9	47.8
343	44.55	50.1	3.9	5.75	89.6	80.4	64.86	7.8	51.1
512	41.72	52.8	3.7	8.92	88.7	78.1	66.71	12.0	55.6
729	37.57	55.7	5.1	5.65	88.8	78.1	64.09	12.3	58.3
10 *processors*									
64	55.83	17.2	22.9	52.28	27.8	22.4	64.11	16.1	35.6
125	38.68	61.3	33.7	44.33	79.8	71.1	49.03	19.8	28.7
216	35.02	64.7	35.0	12.48	81.1	70.2	56.40	9.8	39.4
343	32.15	66.1	35.8	12.76	79.2	68.1	58.40	3.9	44.8
512	26.75	71.7	44.3	15.31	79.6	69.5	58.50	4.5	45.2
729	22.42	75.2	51.8	10.34	82.1	74.6	54.62	7.3	43.1
16 *processors*									

Continued on the next page...

...continued from last page.

	Oxtool			PUB			MPI-2		
q	Efficiency	error (%)		Efficiency	error (%)		Efficiency	error (%)	
	(%)	$const.\ g$	$g(h,h^*)$	(%)	$const.\ g$	$g(h,h^*)$	(%)	$const.\ g$	$g(h,h^*)$
64	63.27	17.6	17.2	55.93	32.2	28.5	78.54	18.7	37.1
125	36.91	60.3	33.5	16.06	75.9	67.7	40.60	29.5	24.1
216	30.20	66.7	41.9	12.13	77.8	71.7	49.18	16.8	29.6
343	24.69	72.6	52.6	10.38	79.7	76.1	47.67	15.3	30.0
512	18.53	78.9	62.2	13.54	81.6	79.0	52.03	9.2	37.0
729	17.08	78.1	61.4	8.28	80.1	79.1	47.40	12.7	32.6
32 *processors*									
125	32.14	57.5	17.4	11.48	66.5	59.9	30.31	24.6	31.6
216	46.73	26.9	15.7	42.93	34.0	33.1	61.52	22.3	35.9
343	24.40	63.4	35.9	10.13	62.7	66.3	39.66	11.5	38.0
512	17.80	73.0	56.5	7.68	69.2	78.6	48.88	22.9	44.6
729	14.76	72.7	55.0	6.94	65.8	76.7	42.65	12.9	38.6

E.2 Results on Distributed Memory, Ethernet (`argus`)

Table E.2: Efficiency and mean relative prediction error on `argus`

	Oxtool			PUB			MPI		
q	Efficiency	rel. error (%)		Efficiency	rel. error (%)		Efficiency	rel. error (%)	
	(%)	$const.\ g$	$g(h,h^*)$	(%)	$const.\ g$	$g(h,h^*)$	(%)	$const.\ g$	$g(h,h^*)$
4 *processors*									
8	36.10	86.4	89.2	-	-	-	-	-	-
64	6.55	38.9	51.6	4.90	35.1	50.0	3.91	25.3	47.9
125	4.63	46.9	59.7	3.98	44.0	57.0	2.92	20.9	48.1
216	3.85	33.8	47.4	2.57	19.6	45.4	2.04	13.9	38.7
343	3.18	43.0	57.9	2.41	32.2	51.1	1.92	9.3	45.5
512	2.49	27.8	46.5	2.11	23.5	42.1	1.40	15.1	34.4
729	2.48	41.9	56.6	1.85	29.7	48.0	1.43	6.0	45.1
10 *processors*									
64	4.04	49.1	69.1	4.37	54.5	70.7	3.99	46.9	66.8
125	2.83	31.2	56.8	1.81	39.2	28.7	2.16	16.8	52.3
216	1.83	32.2	41.0	1.12	57.6	20.5	1.61	25.2	41.8
343	1.93	30.9	48.0	1.00	60.3	24.5	1.70	22.4	46.1
512	1.27	44.5	22.9	0.75	70.3	39.2	1.18	35.9	23.1
729	1.16	40.5	35.1	0.57	72.9	44.2	1.08	29.1	32.5

E.3 Results on Distributed Memory, Myrinet (`aracari`)

Table E.3: Efficiency and mean relative prediction error on `aracari`

	Oxtool			**PUB**			**MPI**		
q	Efficiency	rel. error (%)		Efficiency	rel. error (%)		Efficiency	rel. error (%)	
	(%)	$const.\ g$	$g(h,h^*)$	(%)	$const.\ g$	$g(h,h^*)$	(%)	$const.\ g$	$g(h,h^*)$
4 *processors*									
8	90.30	5.7	14.8	79.99	17.8	12.5	87.70	4.7	10.9
64	63.65	21.4	16.5	21.82	78.7	69.1	62.94	25.2	15.5
125	58.49	22.9	16.5	25.19	71.9	55.1	56.24	26.6	15.0
216	55.64	21.9	21.7	24.15	74.8	55.3	52.56	25.6	17.8
343	51.40	25.2	21.8	24.52	71.0	48.0	46.94	28.6	17.7
512	48.98	28.0	23.3	24.31	72.4	49.5	43.69	28.6	19.2
729	46.64	27.6	26.0	22.30	70.4	43.6	40.91	28.0	20.1
10 *processors*									
64	64.30	16.0	31.3	56.30	11.2	25.0	59.13	13.2	26.6
125	44.21	35.5	8.0	12.54	83.3	75.2	43.61	34.7	12.6
216	42.20	34.0	6.3	8.66	86.6	78.5	40.19	33.9	10.5
343	42.41	29.9	7.3	8.68	85.0	76.3	37.17	30.6	8.1
512	37.34	38.5	8.8	5.87	89.1	82.2	32.88	34.7	11.1
729	33.11	43.0	10.1	6.06	88.5	81.6	28.61	38.8	14.8
16 *processors*									
64	66.19	13.1	26.9	56.34	19.1	10.6	64.29	8.5	20.7
125	41.87	29.4	5.0	11.97	81.8	74.4	41.90	28.9	6.8
216	39.36	31.5	4.9	8.11	85.5	78.3	35.56	35.4	12.3
343	35.36	37.0	9.9	5.03	89.4	84.6	30.98	38.2	17.2
512	29.82	47.2	22.8	3.26	91.8	87.8	27.27	44.3	24.7
729	27.80	44.9	19.7	3.64	90.2	85.8	24.21	42.0	22.6
32 *processors*									
125	33.50	43.9	13.9	11.34	83.9	72.6	33.05	42.4	16.2
216	46.16	14.7	13.4	36.24	57.2	53.8	32.42	27.9	21.6
343	33.14	39.9	8.0	7.42	84.5	76.6	26.04	41.3	19.6
512	28.09	51.8	29.8	4.88	88.3	83.9	19.94	51.1	36.9
729	23.89	50.7	26.5	3.53	89.4	85.2	18.78	47.9	31.2

Appendix F

LLCS Computation — Standard Algorithm

F.1 Results on Shared Memory (`skua`)

Table F.1: Efficiency and mean relative prediction error on `skua`

	Oxtool			**PUB**			**MPI-2**		
α	Efficiency	rel. error (%)		Efficiency	rel. error (%)		Efficiency	rel. error (%)	
	(%)	*const.* g	$g(h)$	(%)	*const.* g	$g(h)$	(%)	*const.* g	$g(h)$
4 *processors*									
1	58.05	3.8	3.8	109.06	29.0	29.1	60.06	1.3	1.3
2	76.10	1.4	1.4	55.98	24.4	24.6	75.40	1.3	1.4
3	81.16	1.5	1.5	57.92	28.0	28.3	80.97	1.2	1.3
4	85.35	1.7	1.6	-	-	-	85.42	1.2	1.5
5	88.21	2.4	2.3	63.10	26.6	27.0	88.11	1.3	1.6
8 *processors*									
1	58.21	2.8	2.7	40.66	30.2	30.0	57.10	3.0	3.1
2	70.39	1.7	1.4	53.42	26.0	25.4	70.21	2.1	2.4
3	78.30	2.1	1.5	56.07	30.8	29.6	78.12	2.4	3.0
4	82.95	2.7	1.8	59.40	31.8	30.1	82.76	3.0	3.9
5	85.96	3.4	2.3	61.50	32.8	30.7	85.82	3.8	5.0
16 *processors*									
1	52.15	1.9	1.3	39.65	23.4	23.1	56.29	4.0	4.0
2	68.62	2.8	1.6	52.19	20.8	20.4	68.31	4.2	4.2
3	76.76	4.1	2.4	66.49	18.9	18.6	75.94	5.7	5.7
4	80.99	4.5	2.2	61.57	17.7	17.4	79.69	7.5	7.5

Continued on the next page...

...continued from last page.

	Oxtool			PUB			MPI-2		
α	Efficiency	error (%)		Efficiency	error (%)		Efficiency	error (%)	
	(%)	$const.\ g$	$g(h)$	(%)	$const.\ g$	$g(h)$	(%)	$const.\ g$	$g(h)$
5	83.03	5.4	2.2	63.37	16.5	16.0	80.91	9.2	9.2
32 *processors*									
1	51.07	4.8	3.2	37.03	27.0	26.0	50.81	3.8	2.7
2	66.53	6.1	3.1	48.01	23.1	20.9	64.87	6.4	3.6
3	71.29	9.5	3.7	51.93	22.4	18.9	67.16	9.2	4.4
4	71.20	12.7	4.4	96.33	20.5	16.4	64.08	12.0	5.9
5	68.17	15.2	4.8	52.64	19.6	14.1	57.34	16.4	9.6

F.2 Results on Distributed Memory, Ethernet (`argus`)

Table F.2: Efficiency and mean relative prediction error on `argus`

	Oxtool			PUB			MPI		
α	Efficiency	rel. error (%)		Efficiency	rel. error (%)		Efficiency	rel. error (%)	
	(%)	$const.\ g$	$g(h)$	(%)	$const.\ g$	$g(h)$	(%)	$const.\ g$	$g(h)$
4 *processors*									
1	80.41	1.7	2.1	80.54	2.2	1.3	79.96	3.3	1.6
2	101.65	2.3	1.7	101.36	6.8	2.2	99.73	2.7	2.7
3	110.35	3.7	2.5	109.15	11.0	3.0	-	-	-
4	115.51	4.3	2.8	114.76	13.9	2.1	109.05	2.1	6.5
5	117.90	5.1	3.0	115.41	17.7	2.2	108.83	1.4	8.8
10 *processors*									
1	70.59	8.1	11.1	71.22	6.7	7.1	67.39	7.3	8.5
2	90.11	10.9	18.3	89.45	12.9	8.0	75.83	4.3	6.8
3	96.80	12.0	23.9	96.44	20.0	8.9	-	-	-
4	97.79	12.5	28.1	97.09	25.8	10.5	54.75	1.8	5.5
5	96.79	13.4	32.3	94.14	30.0	12.1	43.31	1.4	5.4

F.3 Results on Distributed Memory, Myrinet (`aracari`)

Table F.3: Efficiency and mean relative prediction error on `aracari`

	Oxtool			**PUB**			**MPI**		
α	Efficiency	rel. error (%)		Efficiency	rel. error (%)		Efficiency	rel. error (%)	
	(%)	$const.\ g$	$g(h)$	(%)	$const.\ g$	$g(h)$	(%)	$const.\ g$	$g(h)$
4 *processors*									
1	40.63	0.9	0.9	40.05	16.4	16.3	40.33	0.8	0.8
2	51.74	0.8	0.8	51.32	2.4	2.4	51.30	0.9	0.8
3	57.01	0.6	0.6	56.38	2.0	1.9	56.48	0.7	0.6
4	60.14	0.7	0.8	59.40	1.7	1.6	59.54	0.8	0.6
5	62.30	1.0	1.1	61.43	1.7	1.5	61.47	1.0	0.9
8 *processors*									
1	37.15	2.4	2.2	36.81	9.6	9.3	36.95	2.7	2.7
2	48.80	1.1	1.6	48.44	2.5	1.7	48.38	1.6	1.7
3	55.38	1.8	3.0	54.18	2.2	1.0	53.74	0.9	1.0
4	58.92	2.6	4.3	57.45	2.5	0.8	56.82	0.6	0.7
5	61.24	3.1	4.9	59.69	3.0	0.7	58.63	0.9	1.0
16 *processors*									
1	35.43	2.9	3.4	35.26	4.5	4.1	35.06	3.9	3.3
2	47.23	1.2	3.7	46.98	3.2	3.0	45.83	2.6	1.2
3	53.90	2.7	6.4	52.60	3.8	3.4	49.64	2.7	0.6
4	56.83	3.2	8.2	55.27	4.8	4.3	49.63	2.7	0.3
5	58.17	3.6	10.7	56.43	5.7	5.5	47.73	3.3	0.3
32 *processors*									
1	34.45	2.6	5.2	34.36	4.3	5.3	33.08	4.4	2.2
2	45.31	2.5	11.3	45.11	3.1	9.4	38.06	3.9	1.1
3	48.91	3.9	16.1	48.35	3.6	14.7	33.26	4.5	1.4
4	47.57	3.6	18.8	48.24	4.3	18.8	26.83	4.5	2.2
5	43.43	2.9	20.1	45.99	5.9	21.1	20.92	5.3	1.9

Appendix G

LLCS Computation — Bit-Parallel Algorithm

G.1 Results on Shared Memory (`skua`)

Table G.1: Efficiency and mean relative prediction error on `skua`

	Oxtool			**PUB**			**MPI-2**		
α	Efficiency	rel. error (%)		Efficiency	rel. error (%)		Efficiency	rel. error (%)	
	(%)	$const.\ g$	$g(h)$	(%)	$const.\ g$	$g(h)$	(%)	$const.\ g$	$g(h)$
4 *processors*									
1	80.06	29.1	29.1	85.18	30.7	30.7	79.46	29.2	29.3
2	83.98	32.2	32.2	87.81	31.4	31.5	83.07	32.2	32.2
3	85.00	38.7	38.7	80.57	37.3	37.4	84.65	39.2	39.2
4	71.11	49.9	49.9	73.63	47.6	47.8	69.43	50.4	50.5
5	70.30	57.7	57.7	67.49	56.7	56.9	69.95	57.8	58.0
8 *processors*									
1	60.10	33.4	33.4	64.18	32.5	32.5	60.35	32.5	32.6
2	58.03	52.8	52.8	60.24	51.2	51.1	57.45	53.3	53.5
3	53.80	62.0	62.1	52.71	61.5	61.5	53.63	62.0	62.3
4	46.54	68.6	68.6	45.96	69.1	69.1	44.05	69.9	70.2
5	41.18	75.1	75.1	40.85	75.4	75.3	40.78	82.8	83.7
16 *processors*									
1	42.48	53.8	53.8	44.71	51.1	51.2	42.39	53.7	53.8
2	37.64	70.2	70.2	37.89	69.2	69.4	37.49	69.9	70.3
3	30.38	77.4	77.5	30.50	76.1	76.5	30.23	77.2	77.6
4	24.97	81.7	81.8	25.82	80.2	80.7	24.61	81.4	82.0

Continued on the next page...

...continued from last page.

	Oxtool			PUB			MPI-2		
α	Efficiency	error (%)		Efficiency	error (%)		Efficiency	error (%)	
	(%)	$const.\ g$	$g(h)$	(%)	$const.\ g$	$g(h)$	(%)	$const.\ g$	$g(h)$
5	21.78	84.4	84.5	22.23	82.8	83.5	21.43	83.7	84.5
32 *processors*									
1	27.33	71.0	71.0	27.95	70.3	70.6	27.11	70.8	70.9
2	20.42	81.8	81.8	21.30	80.6	81.5	20.16	80.3	80.6
3	16.30	84.9	84.9	16.80	84.1	85.2	16.07	82.8	83.2
4	13.31	86.6	86.5	13.79	85.6	87.0	13.01	83.8	84.3
5	11.41	86.5	86.5	11.68	85.4	87.4	11.16	82.8	83.5

G.2 Results on Distributed Memory, Ethernet (`argus`)

Table G.2: Efficiency and mean relative prediction error on `argus`

	Oxtool			PUB			MPI		
α	Efficiency	rel. error (%)		Efficiency	rel. error (%)		Efficiency	rel. error (%)	
	(%)	$const.\ g$	$g(h)$	(%)	$const.\ g$	$g(h)$	(%)	$const.\ g$	$g(h)$
4 *processors*									
1	56.09	2.0	2.8	56.34	3.4	2.5	56.11	2.9	2.1
2	71.43	2.7	1.5	71.56	6.5	1.5	70.60	1.5	3.0
3	79.27	3.9	1.6	79.35	10.4	2.0	77.53	1.0	5.0
4	83.35	6.5	2.8	84.08	14.0	2.4	80.45	1.8	7.3
5	86.33	8.5	3.5	86.44	17.1	2.9	81.93	2.8	9.5
10 *processors*									
1	47.93	8.9	11.0	48.17	8.8	7.1	46.95	7.4	8.3
2	64.12	4.3	9.7	64.19	12.5	3.8	57.77	2.8	4.3
3	71.25	1.7	9.5	70.33	20.0	3.5	-	-	-
4	74.12	2.4	10.4	72.66	25.9	4.2	50.92	1.5	1.7
5	74.26	2.5	13.3	71.25	30.4	5.1	42.90	1.9	1.7

G.3 Results on Distributed Memory, Myrinet (`aracari`)

Table G.3: Efficiency and mean relative prediction error on `aracari`

	Oxtool			**PUB**			**MPI**		
α	Efficiency	rel. error (%)		Efficiency	rel. error (%)		Efficiency	rel. error (%)	
	(%)	$const.\ g$	$g(h)$	(%)	$const.\ g$	$g(h)$	(%)	$const.\ g$	$g(h)$
4 *processors*									
1	56.60	1.8	1.9	56.54	14.6	14.6	56.62	1.8	1.8
2	72.06	1.6	1.8	71.96	10.0	10.1	72.06	1.8	1.8
3	79.57	3.1	4.0	79.74	2.3	2.5	79.83	1.3	1.3
4	84.51	1.1	1.4	84.23	2.1	2.3	84.36	1.3	1.3
5	87.58	1.3	1.7	87.29	2.6	2.9	87.38	1.7	1.6
8 *processors*									
1	50.51	6.0	6.1	50.46	12.5	12.5	50.51	5.8	6.0
2	67.46	3.8	4.0	67.34	4.7	4.8	67.27	6.1	6.4
3	73.88	5.2	6.2	75.65	4.7	4.9	75.52	3.2	3.8
4	81.11	2.2	2.7	80.68	4.7	4.9	80.34	2.7	3.5
5	84.28	2.2	2.7	83.77	5.6	5.8	83.03	2.8	3.9
16 *processors*									
1	46.60	15.8	16.1	47.57	9.4	9.4	47.57	8.7	8.8
2	64.86	5.4	5.6	64.65	7.4	7.4	63.84	7.0	7.2
3	70.77	5.6	6.2	72.81	7.7	7.7	70.68	7.0	7.3
4	77.96	3.5	3.9	77.36	7.8	7.8	72.96	7.1	7.5
5	80.61	4.2	4.8	79.85	9.6	9.6	73.43	5.7	6.2
32 *processors*									
1	45.98	8.3	8.4	45.91	9.9	9.8	45.14	9.9	9.9
2	62.04	4.1	4.4	61.97	7.9	7.8	56.01	8.0	8.1
3	68.56	3.2	3.6	68.51	7.8	7.7	56.07	6.1	6.2
4	70.40	2.3	2.8	70.74	8.3	8.1	50.65	5.7	5.9
5	47.85	3.9	4.8	70.65	9.5	9.3	43.82	6.0	6.2

Bibliography

[1] A. Alexandrov, M. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model – one step closer towards a realistic model for parallel computation. In *Proc. 7th Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'95*, pages 95—105, Santa Barbara, California, 1995. UC Santa Barbara.

[2] Lloyd Allison and Trevor I. Dix. A bit-string longest-common-subsequence algorithm. *Inf. Process. Lett*, 23(6):305—310, 1986.

[3] C. E. R. Alves, E. N. Cáceres, and F. Dehne. Parallel dynamic programming for solving the string editing problem on a CGM/BSP. In *Proc. of the 14th Annual ACM symposium on Parallel Algorithms and Architectures (SPAA '02)*, pages 275—281, New York, NY, USA, 2002. ACM Press.

[4] Alberto Apostolico, Mikhail J. Atallah, Lawrence L. Larmore, and Scott McFaddin. Efficient parallel algorithms for string editing and related problems. *SIAM J. Comput.*, 19(5):968—988, 1990.

[5] M. Bianco and G. Pucci. On the Predictive Quality of BSP-like Cost Functions for NOWs. *Proc. of Euro-Par 2000, Lecture Notes in Computer Science*, 1900:638—646, 2000.

[6] G. Bilardi, C. Fantozzi, A. Pietracaprina, and G. Pucci. On the Effectiveness of D-BSP as a Bridging Model of Parallel Computation. *Proc. ICCS '01, Lecture Notes in Computer Science*, 2074:579—588, 2001.

[7] G. Bilardi, K. T. Herley, A. Pietracaprina, G. Pucci, and P. Spirakis. BSP versus LogP. *Algorithmica*, 24(3-4):405—422, 1999.

[8] R.H. Bisseling. `http://www.math.uu.nl/people/bisseling/software.html`.

[9] R.H. Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP and MPI*. Oxford University Press, 2004.

[10] V. Blanco, J. A. Gonzalez, C. Leon, C. Rodriguez, G. Rodriguez, and M. Printista. Predicting the performance of parallel programs. *Parallel Computing*, 30(3):337—356, 2004.

[11] Olaf Bonorden, Joachim Gehweiler, and Friedhelm Meyer auf der Heide. A Web Computing Environment for Parallel Algorithms in Java. In *Proceedings of International Conference on Parallel Processing and Applied Mathematics (PPAM)*, September 2005. to appear.

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

[13] A. Chan and F.K.H.A. Dehne. CGMgraph/CGMlib: Implementing and Testing CGM Graph Algorithms on PC Clusters. In *Proc. of EuroPVM/MPI, LNCS 2840*, pages 117—125, 2003.

[14] J. Choi, Jack J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. A proposal for a set of parallel basic linear algebra subprograms. LAPACK Working Note 100, University of Tennessee, 1995.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2nd edition, 2001.

[16] Maxime Crochemore, Costas S. Iliopoulos, and Yoan J. Pinzon. Recovering an LCS in $O(n^2/w)$ time and space. *Colombian Journal of Computation*, 3(1):41–52, 2002.

[17] Maxime Crochemore, Costas S. Iliopoulos, and Yoan J. Pinzon. Speeding-up Hirschberg

and Hunt–Szymanski algorithms for the LCS problem. *Fundamenta Informaticae*, 56(1–2):89—103, 2003.

[18] Maxime Crochemore, Costas S. Iliopoulos, Yoan J. Pinzon, and James F. Reid. A fast and practical bit-vector algorithm for the Longest Common Subsequence problem. *Information Processing Letters*, 80(6):279—285, December 2001.

[19] 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. *Commun. ACM*, 39(11):78—85, 1996.

[20] Pilar de la Torre and Clyde P. Kruskal. Submachine Locality in the Bulk Synchronous Setting. *Lecture Notes in Computer Science*, 1124:352—358, 1996.

[21] Jack J. Dongarra. Performance of various computers using standard linear equations software in a FORTRAN environment. *ACM SIGARCH Comput. Archit. News*, 16(1):47—69, 1988.

[22] Jack J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1—17, March 1990.

[23] Jack J. Dongarra, C. B. Moler, J. R. Bunch, and George W. Stewart. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1979.

[24] M. Essaidi, I. Guérin Lassous, and J. Gustedt. SSCRAP: An Environment for Coarse Grained Algorithms. In S.G. Akl et al., editor, *14th IASTED International Conference on Parallel and Distributed Computing and Systems – PDCS'2002, Boston, MA, USA*, pages 398—403. IASTED, November 2002.

[25] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2003.

[26] Thierry Garcia, Jean Frédéric Myoupo, and David Semé. A coarse-grained multicomputer algorithm for the longest common subsequence problem. In *Euro PDP*, pages 349—356, 2003.

[27] A. Geist, A. Beguelin, Jack J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, Mass., 1994.

[28] A. Gerbessiotis and F. Petrini. Network performance assessment under the BSP model. In *Proc. International Workshop on Constructive Methods for Parallel Programming, Marstrand/Gothenborg, Sweden*, 1998. An extended version of this work is also available as Technical Report PRG-TR-03-98, Computing Laboratory, Oxford Unversity, 1998.

[29] A. V. Gerbessiotis and S. Y. Lee. Remote memory access: A case for portable, efficient and library independent parallel programming. *Scientific Programming*, 12(3):169—184, 2004.

[30] Alan Gibbons and Wojciech Rytter. *Efficient parallel algorithms*. Cambridge University Press, New York, NY, USA, 1988.

[31] Andrei Goldchleger, Fabio Kon, Alfredo Goldman, Marcelo Finger, and Germano Capistrano Bezerra. InteGrade: Object-Oriented Grid middleware leveraging the idle computing power of desktop machines. *Concurrency and Computation: Practice and Experience*, 16(5):449—459, 2004.

[32] Brian Grayson, Michael Dahlin, and Vijaya Ramachandran. Experimental evaluation of QSM, a simple shared-memory model. In *Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, IPPS/SPDP'99 (San Juan, Puerto Rico, April 12-16, 1999)*, pages 130—136, Los Alamitos-Washington-Brussels-Tokyo, 1999. IEEE Computer Society, ACM SIGARCH, IEEE Computer Society Press.

[33] Douglas Gregor and Andrew Lumsdaine. The parallel BGL: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[34] W. D. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and

M. Snir. *MPI: The Complete Reference. Volume 2, The MPI-2 Extensions*. Scientific and Engineering Computation. MIT Press, Cambridge, MA, USA, second edition, 1998.

[35] W. D. Gropp and E. Lusk. *Installation Guide for* `mpich`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

[36] W. D. Gropp and E. Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

[37] J. M. D. Hill, P. I. Crumpton, and D. A. Burgess. The theory, practice, and a tool for BSP performance prediction applied to a CFD application. Technical report PRG-TR-4-96, Programming Research Group, Oxford University Computing Laboratory, 1996.

[38] J. M. D. Hill, W. F. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. BSPlib: The BSP Programming Library. *Parallel Computing*, 24(14):1947—1980, May 1998.

[39] J. M. D. Hill and D. B. Skillicorn. Lessons learned from implementing BSP. *Future Generation Computer Systems*, 13(4–5):327—335, 1998.

[40] J. M. D. Hill and D. B. Skillicorn. Practical barrier synchronisation. In *6th EuroMicro Workshop on Parallel and Distributed Processing (PDP'98)*. IEEE Computer Society Press, January 1998.

[41] Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341—343, 1975.

[42] Guy Horvitz and R. H. Bisseling. Designing a BSP version of ScaLAPACK. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[43] Sascha Hunold, Thomas Rauber, and Gudula Rünger. Multilevel hierarchical matrix multiplication on clusters. In *ICS '04: Proceedings of the 18th Annual International Conference on Supercomputing*, pages 136—145, New York, NY, USA, 2004. ACM Press.

[44] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350—353, May 1977.

[45] Heikki Hyyrö. Bit-parallel LCS-length computation revisited. In *Proc. 15th Australasian Workshop on Combinatorial Algorithms (AWOCA 2004)*, 2004.

[46] Heikki Hyyrö, Jun Takaba, Ayumi Shinohara, and Masayuki Takeda. On bit-parallel processing of multi-byte text. In *Asia Information Retrieval Symposium, AIRS 2004, Beijing, China, Lecture Notes in Computer Science*, volume 3411, pages 289—300, 2005.

[47] Ben Juurlink and Ingo Rieping. Performance relevant issues for parallel computation models. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, July 13 2001.

[48] Ben H. H. Juurlink and Harry A. G. Wijshoff. The E-BSP Model: Incorporating General Locality and Unbalanced Communication into the BSP Model. In *Euro-Par '96: Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II, LNCS 1124*, pages 339—347, London, UK, 1996. Springer-Verlag.

[49] Ben H. H. Juurlink and Harry A. G. Wijshoff. A quantitative comparison of parallel computation models. *ACM Trans. Comput. Syst.*, 16(3):271—318, 1998.

[50] Peter Krusche. Experimental evaluation of BSP programming libraries. In *Draft proceedings of HLPP 2005 workshop*, 2005. submitted to PPL.

[51] Peter Krusche and A. Tiskin. Efficient Longest Common Subsequence Computation using Bulk-Synchronous Parallelism. In *Proceedings of ICCSA 2006, LNCS 3984*, 2006.

[52] Qingshan Luo and John B. Drake. A scalable parallel Strassen's matrix multiplication algorithm for distributed-memory computers. In *SAC '95: Proceedings of the 1995 ACM Symposium on Applied Computing*, pages 221—226, New York, NY, USA, 1995. ACM Press.

[53] Jeremy M. R. Martin and A. Tiskin. Dynamic BSP: Towards a Flexible Approach to Parallel Computing over the Grid. In Ian R. East, David Duce, Mark Green, Jeremy

M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, pages 219—226, 2004.

[54] Hakan Mattsson and Christoph W. Kessler. Towards a virtual shared memory programming environment for grids. In *Proceedings of PARA'04, 2004, Lecture Notes in Computer Science*, to appear.

[55] W. F. McColl and A. Tiskin. Memory-efficient matrix computations in the BSP model. *Algorithmica*, 24(3–4):287—297, 1999.

[56] Panagiotis D. Michailidis and Konstantinos G. Margaritis. New processor array architectures for the longest common subsequence problem. *The Journal of Supercomputing*, 32(1):51—69, 2005.

[57] R. Miller. *Two Approaches to Architecture-Independent Parallel Computation*. PhD thesis, Oxford University Computing Laboratory, Michaelmas Term 1994.

[58] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31—88, 2001.

[59] Vijaya Ramachandran. QSM: A General Purpose Shared-Memory Model for Parallel Computation. *Lecture Notes in Computer Science*, 1346:1—5, 1997.

[60] D.B. Skillicorn, J.M.D. Hill, and W.F. McColl. Questions and Answers about BSP. *Scientific Computing*, 6(3):249—274, 1997.

[61] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, and Jack J. Dongarra. *MPI: The Complete Reference. Volume 1, The MPI-1 Core*. MIT Press, Cambridge, MA, USA, second edition, September 1998. See also volume 2 [34].

[62] A. Tiskin. Bulk-synchronous parallelism: An emerging paradigm of high-performance computing. In L. T. Yang and M. Guo, *High Performance Computing: Paradigm and Infrastructure*, pp. 69 — 80, John Wiley and Sons, 2005.

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

[64] R. van de Geijn and J. Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255—274, 1997.

[65] A. Zavanella. The Skel-BSP Global Optimizer: Enhancing Performance Portability in Parallel Programming. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 658—667, London, UK, 2000. Springer-Verlag.

[66] A. Zavanella. Skeletons, BSP and performance portability. *Parallel Processing Letters*, 11(4):393—407, 2001.

[67] A. Zavanella and A. Milazzo. Predictability of Bulk Synchronous Programs Using MPI. In *8th Euromicro Workshop on Parallel and Distributed Processing,*, pages 118—?, 2000.

[68] ATLAS: `http://math-atlas.sourceforge.net/`.

[69] CGMLib: `http://www.scs.carleton.ca/~cgm/`.

[70] LAM/MPI: `http://www.lam-mpi.org/`.

[71] MPICH: `http://www-unix.mcs.anl.gov/mpi/mpich/`.

[72] Myricom, Inc.: `http://www.myri.com/`.

[73] Netlib/BLAS: `http://www.netlib.org/blas/`.

[74] Netlib/SCALAPACK: `http://www.netlib.org/scalapack/`.

[75] PUB library: `http://www.uni-paderborn.de/~bsp/`.

[76] The Oxford BSP Toolset: `http://www.bsp-worldwide.org/implmnts/oxtool/`.

[77] The GNU C Compiler: `http://www.gnu.org/software/gcc/`.

[78] Gnuplot: `http://www.gnuplot.info`.

[79] Intel corp.: `http://www.intel.com`.

[80] Spec: `http://www.spec.org`.