EXPERIMENTAL EVALUATION OF
BSP PROGRAMMING LIBRARIES

PETER KRUSCHE

Department of Computer Science/Centre for Scientific Computing, University of Warwick
Coventry, CV4 7AL, United Kingdom

ABSTRACT
The model of bulk-synchronous parallel computation (BSP) helps to implement portable general purpose algorithms while keeping predictable performance on different parallel computers. Nevertheless, when programming in ‘BSP style’, the running time of the implementation of an algorithm can be very dependent on the underlying communications library. In this study, an overview of existing approaches to practical BSP programming in C/C++ or Fortran is given and benchmarks were run for the two main BSP-like communications libraries, BSPlib and PUB. Furthermore, a memory efficient matrix multiplication algorithm was implemented and used to compare their performance on different parallel computers and to evaluate the compliance with predictions by theoretical results.

1. Introduction
Bulk synchronous parallelism is a model for parallel programming which achieves scalability and portability between different parallel architectures as well as realistic theoretical modeling of distributed memory systems. A BSP computer consists of a set of processor-memory pairs, which communicate over an arbitrary interconnection network. The programs are SPMD, low level program structures are divided into supersteps and it is ensured that all communication is finished by the start of the next superstep. In this way, the analysis of communication and computation performance can be separated. The upper and lower bounds for the running time of a BSP program on an arbitrary parallel computer can thus be estimated theoretically by knowing the parameters of its communication network and the computation speed of a single processor.

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 evaluate different approaches to practical BSP programming and compare their performance on different parallel computers as well as the resemblance to theoretical results.

*email: P.Krusche@warwick.ac.uk*
2. Approaches to BSP Programming

A first approach to BSP programming in C/C++ or Fortran might be to use general communication libraries like MPI or PVM [17,10] and write programs in ‘BSP style’. An example is given by Bisseling in [3] and on the corresponding website [3] as project MPIedupack.

Another approach is to use an implementation of BSPlib, which defines a library interface for direct BSP programming [13,16]. It includes the main primitives for synchronization, remote memory access and bulk synchronous message passing. Its aim is to simplify BSP programming and keep platform independence by introducing a common standard. It also allows the implementation to have more advanced optimization methods, like optimized barrier synchronization, combining messages to the same destination [14] 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), which has been compiled and tested on various architectures and systems [24]. More importantly, it has been optimized for message passing, native DRMA and shared memory platforms – which is particularly useful as on some shared memory/DRMA platforms, emulated message passing can lead to additional overhead.

The other implementation is the PUB library from Paderborn University [23], which includes some features not present in the BSPlib standard [6], such as oblivious synchronization and partitioning the processors into subgroups (which can also yield better synchronization performance). The authors of PUB argue that the BSP model is not sufficient for realistic performance modeling of simple communications primitives (message passing/remote memory access) and thus they also provide advanced communication functionality, e.g. for broadcasting and reducing data.

In [11] the performance of the remote memory access functions of various communication libraries (Oxtool, PUB and LAM-MPI) is compared for different implementations of algorithms. Also, a framework for communications-library independent BSP like programming is presented there. Other studies on the predictability of BSP algorithms are contained in [1,25] (using MPI) and [5] (using MPI and PUB).

Another related project is CGMlib [20], an implementation of CGM (a model for parallel programming similar to BSP) which runs on top of MPI. CGMlib provides a high level programming interface in C++ for communicating lists of abstract datatypes with constant size. This differs from the BSPlib approach, in which messages contain arrays of bytes which can have variable length. CGMlib also provides implementations of advanced communication primitives (broadcast, reduction, h-Relation). There are implementations of various standard algorithms like sorting and parallel prefix, as well as of several CGM graph algorithms for this library (the CGMGraph library - to be found at the same website).

Finally there is a project called SSCRAP (or Soft Synchronized Computing in Rounds for Adequate Parallelization) [9], which provides a C++ library that contains primitives for synchronization, communication and processor group management. It runs on top of MPI or Posix Threads. Algorithms for list ranking, sorting and prefix sums are also available.
A distinction between these libraries can be made by comparing the communication primitives they provide. Simple communication primitives are direct synchronous/bulk synchronous message passing (MP/BSMP) and direct remote memory access (DRMA). A short comparison between these features is shown in Table 1, a more extensive list can be found in [9]. Bisseling also gives a summary about different BSP programming environments and shows benchmark results for various parallel computers in [4].

<table>
<thead>
<tr>
<th>Feature</th>
<th>Oxtool</th>
<th>PUB</th>
<th>CGMlib</th>
<th>SSCRAP</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP</td>
<td>-</td>
<td>-</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>BSMP</td>
<td>•</td>
<td>•</td>
<td>-</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>DRMA</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

Based on this comparison, Oxtool and PUB have been chosen for a first closer evaluation. A C++ wrapper library which provides a front end to BSMP, DRMA, broadcasting and reduction operation was created and used to compare their performance with 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 will still be used for the experiments here because it is portable to all of the used machines. In particular, one of the machines used for the experiments has a Myrinet interconnection network which is 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. An MPI implementation that provides BSPlib style functionality using non-blocking send/receive operations was created as well. This implementation presumably has less overhead than BSPlib or PUB, but does not include facilities for message combining or splitting and other possible methods of optimizing performance.

CGMlib has no BSMP or DRMA facilities, and it is based on C/C++ datatypes instead of buffers of bytes. Because of this difference in programming models, CGMlib was not taken into consideration for the experiments that followed.

Comparing the libraries makes sense because there are various differences in the actual primitive implementations, which can lead to differences in performance on the various platforms.

3. BSP Performance Evaluation

The BSP computer was introduced in [18], its main ingredients being a set of processor/memory pairs that are connected by a communication network. Program execution takes place in supersteps, each consisting of an input phase, a set of local computations and an output phase. Between these supersteps, a barrier-style synchronization takes place. Following this scheme, it is sufficient to compute the running time for each superstep and then the sum over all supersteps to get the total running time.

The base unit for computation cost is the time needed for a simple arithmetic operation or a memory access. Let $w$ be the maximum number of local operations, $h'$ the maximum number of data units received and $h''$ the maximum number of
data units sent by each processor during a specific superstep.

Furthermore, let a single processor be able to do a local operation in a fixed time $f$. The inverse $1/f$ 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 throughput describes how fast data can be transmitted by the network once a transfer of a message has started. The communication latency $l$ can be seen as the time needed for overhead caused by starting up communication.

For an algorithm consisting of $S$ supersteps, the cost of superstep $s$ is defined as $C_s = f \cdot w_s + g \cdot h_s + l$ with $h_s = h'_s + h''_s$, the total cost is given by the sum $C = \sum_{s=1}^{S} C_s = f \cdot W + g \cdot H + l \cdot S$, where $W = \sum_{s=1}^{S} w_s$ and $H = \sum_{s=1}^{S} h_s$.

The parameters $f$, $l$ and $g$ can be determined before running the program by using a modified version of the benchmarking tool $bspprobe$ which was originally supplied with the Oxford BSP toolset.

$bspprobe$ in its original version measures $g$ for two kinds of communication: a local shift, where each processor sends data to exactly one other, and all-to-all, where each processor sends data to all the others. It also takes samples for different message sizes. Ideally $g$ should be constant overall; in practice there will be overhead for small messages that increases $g$ when the total communication size is small. The problem is that $g$ depends on the message size and the number of messages that is sent per superstep, e.g. depending on how well the message combining functionality of the communications library works on a particular message passing system. Also, the performance can be dependent on the communication primitive that is used, cf. e.g. [11]. Because of this, the performance was measured with a different benchmarking routine, which takes samples for different numbers of messages and message sizes. The communication pattern of the algorithm decides from which area of the resulting surface of bandwidth values a value of $g$ for a good performance estimation can be taken.

The parameter $l$ is measured with and without communication (high and low): $bspprobe$ in particular only measures the latency for a local shift operation. Again, in [11] it is argued that this is not sufficient, since e.g. all-to-all communication can create a much higher latency value. Because of this, the latency for an all-to-all communication was measured as well.

Another approach for getting the parameters $g$ and $l$ is to measure communication times for different numbers of messages and then doing a linear regression for this data (see Bisseling [4]). If the time that is used for doing the communication increases approximately linear to the number of messages, the slope

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig1.png}
\caption{Matrix multiplication — comparison of local(sequential) performance}
\end{figure}
of the fitted line will give \( g \) and the time at its intersection with the time-axis will give \( h \).

\text{Bpprobe} also measures two different approximations of the flop rate \( 1/f \), which are included here because they give a good algorithm-independent estimation. The measurement functions for the local computation speed was changed to use BLAS \cite{18,22}, which is less dependent on local cache sizes and will provide good performance on every platform. To illustrate this further, Fig. 1 shows the performance of two different sequential dense matrix multiplication kernels on a specific platform (in a later section it will be referred to as \text{aracari}). One uses a simple \( i, j, k \) loop that computes the matrix product in \( O(n^3) \) time, the other one uses an optimized level 3 BLAS implementation. An efficient implementation for the local computation phase leads to better predictability, as the performance of the ATLAS/BLAS version has less dependency on cache and problem sizes. Nevertheless, if possible the value of \( f \) should be determined more accurately by running the algorithms under consideration on just one processor first and measuring time for the computation phase separately.

4. Benchmarking Experiments

The first experiments were done on two different parallel machines at the Centre for Scientific Computing at the University of Warwick. The first one (\text{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 node). It offers very good communication performance, but has slower individual processors. The second one (\text{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 node), it has better local computation performance but a slower communications network (100 Mbit Ethernet). On both systems, the libraries were compiled on top of MPI. The experiments up to now used 1 to 32 processors on \text{aracari} due to the job queueing system which can create long waiting times for jobs that request many nodes at once. For similar reasons, the tests on \text{argus} had to be restricted to 4 nodes, particularly because this system is mostly used for task farming and jobs that use few processors. Getting the same nodes in the cluster cannot always be guaranteed, and furthermore, the traffic on the communications network can be different every time because other users are also running jobs on different nodes. Nevertheless, these conditions provide a realistic testing environment and can show rather well how differently optimized communication libraries perform.

For all the benchmarking measurements, 10 samples were taken and averaged to get reliable results. For testing communication performance, units of 8-byte double values were transmitted, because the same data type will be used for all local computations later on. In the following, all the ‘flops’ will also correspond to double precision floating point operations and all the message sizes are in units of double values.

The value of \( f \) was estimated by fitting to the local computation times of the matrix multiplication algorithm that will be studied in the following sections. Fig. 2 shows the results for the local computation time and the theoretical values estimated with \( f \) (\text{dgemm}) from Table 2.

The values of \( l \) were measured for three types of communication: \( l \) (low) is the
Table 2. values of $f$ and $l$

<table>
<thead>
<tr>
<th></th>
<th>MPI</th>
<th>Oxtool</th>
<th>PUB</th>
<th>MPI</th>
<th>Oxtool</th>
<th>PUB</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f$ (average)</td>
<td>2.7 ns/flop</td>
<td>1.9 ns/flop</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$f$ (dgemm)</td>
<td>2.5 ns/flop</td>
<td>0.83 ns/flop</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>aracari 4 processors</th>
<th>argus 4 processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l$ (low)</td>
<td>210 µs</td>
<td>43 µs</td>
</tr>
<tr>
<td>$l$ (high)</td>
<td>230 µs</td>
<td>67 µs</td>
</tr>
<tr>
<td>$l$ (all-to-all)</td>
<td>252 µs</td>
<td>89 µs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>aracari 10 processors</th>
<th>aracari 16 processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l$ (low)</td>
<td>776 µs</td>
<td>168 µs</td>
</tr>
<tr>
<td>$l$ (high)</td>
<td>792 µs</td>
<td>190 µs</td>
</tr>
<tr>
<td>$l$ (all-to-all)</td>
<td>942 µs</td>
<td>335 µs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>aracari 32 processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l$ (low)</td>
<td>2203 µs</td>
</tr>
<tr>
<td>$l$ (high)</td>
<td>2242 µs</td>
</tr>
<tr>
<td>$l$ (all-to-all)</td>
<td>2881 µs</td>
</tr>
</tbody>
</table>

Fig. 2. Matrix multiplication — local computation times

As can be seen from Fig. 2, the value of $g$ is dependent on both the number and size of the messages which are transmitted per superstep. Note that the scale in these figures is logarithmic, let $m$ denote the number of messages that is transmitted and $s$ the size in 8-byte double units. In the BSP model this surface would ideally just contain constant values. In practice it has a slope which is also non-homogeneous in the number- and message size direction. In particular, for PUB the time needed to transfer an element becomes larger again when the number of messages increases beyond a certain limit. For small messages, MPI is slower than
the other libraries, for large/many messages Oxtool and MPI are leading. The best performance on average is achieved by Oxtool which is only slightly slower than for large messages MPI. In Fig. 3 the performance of PUB, Oxtool and MPI are shown as opaque surfaces, so only the upper envelope is visible. The hidden part below the upper envelope shows that for a small message size and varying message counts the performance remains approximately constant for PUB and Oxtool due to their message combining facilities, whereas it decreases for MPI. On argus, all the libraries show similar performance, although there are slight advantages for Oxtool. On aracari, the performance of MPI and Oxtool is similar for large messages on up to 16 processors, on 32 processors the values of $g$ measured for a certain block size become larger for MPI, whereas they remain approximately constant for Oxtool. For all numbers of processors on aracari, PUB shows a decrease in performance, once a certain communication volume is reached, as was also observed in [11]. A difference can be seen on aracari for 32 processors, there the performance of the MPI version decreases for large buffer sizes.

Tbl. 3 contains the values of $g$ for sending square matrices of the specified width row by row. Each row is assumed to be transferred separately, giving the $(m, s)$ pairs $(16, 8), (32, 16), ...$ for matrix sizes 8, 16, ... to resemble the communication pattern of the matrix multiplication algorithm in the following section.

5. Dense Matrix-Matrix Multiplication

As a practical benchmark, a memory efficient dense matrix-matrix multiplication algorithm has been implemented. A simple sequential algorithm for computing the matrix product

$$C = A \cdot B$$

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

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

with $i, j = 1, 2, ..., n$. This is equivalent to representing the elementary products as points of a $n \times n \times n$ cube, with data broadcasting in two dimensions and
Table 3. values of $g$ in $\mu$s

<table>
<thead>
<tr>
<th>Mat. width</th>
<th>MPI Oxtool PUB</th>
<th>argus, 4 processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>aracari, 4 processors</td>
<td>32</td>
<td>0.74</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>0.45</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>0.36</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>0.34</td>
</tr>
<tr>
<td>aracari, 10 processors</td>
<td>32</td>
<td>1.54</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>0.70</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>0.58</td>
</tr>
<tr>
<td>aracari, 16 processors</td>
<td>32</td>
<td>2.69</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>1.90</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>1.95</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>3.09</td>
</tr>
</tbody>
</table>

Data combining by addition in the third dimension. The BSP algorithm used here is described in [15], the basic idea is to partition the problem into cubic blocks, which correspond to products of smaller matrices that can then be computed locally on each processor. The smaller these blocks are, the less temporary storage per processor will be used. A smaller block size increases the number of supersteps that are necessary and decreases the communication volume per superstep. The communication pattern of the algorithm can thus be controlled by modifying the block size parameter, which is why it seems to be a good choice for studying the differences in performance of BSP-style communication library implementations.

The matrix multiplication cube will be partitioned into a number $q > p$ (for keeping memory efficiency) of blocks $V_{IJK}$ with $1 \leq I, J, K \leq q^{1/3}$. These blocks will then be processed locally on each processor, using an optimized multiplication kernel as the level 3 BLAS routine $\text{dgemm}$. The blocks $V_{IJK}$ are defined as

$$V_{IJK} = A_{IJ} \cdot B_{JK} \quad \text{with} \quad I, J, K = 1, 2, \ldots, q^{1/3} \quad (3)$$

$A_{IJ}$ and $B_{JK}$ are the 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 size $s \times s$ with block size $s = n/q^{1/3}$.

The temporary results $V_{IJK}$ will then be added to obtain the result matrix $C$:

$$C_{IK} = \sum_{J=1}^{q^{1/3}} V_{IJK} \quad \text{with} \quad (I, K = 1, 2, \ldots, q^{1/3}) \quad (4)$$

The input and output data are distributed in a 2-dimensional grid of blocks with size $n/\sqrt{p} \times n/\sqrt{p}$. The BSP running time of this algorithm is shown in Eq. (5),

$$T = f \cdot \left\lceil \frac{q}{p} \right\rceil n^3 \cdot \frac{q}{q} + g \cdot \left\lceil \frac{q}{p} \right\rceil n^2 \cdot \left(2 + \frac{1}{q^{1/3}}\right) + l \cdot \left(2 \left\lceil \frac{q}{p} \right\rceil \right)$$

(5)
The factor of $2 + 1/q^{1/3}$ when estimating $T$ (Eq. 5) is composed of the input phase, where two submatrices have to be sent, and of the output phase. Adding the values in $V_{IJK}$ to their final position in $C$ has to be done only every $q^{1/3}$ supersteps because the results can be added up and buffered locally if every processor computes its own block column $V_{I,(1,\ldots,q^{1/3}),K}$. The input data still has to be distributed in every superstep to retain memory efficiency.

For transmitting the parts of $A$ and $B$, the DRMA facilities of the libraries will be used, and when $\sqrt{p} \neq q^{1/3}$, the matrices will be transferred line by line. This leads to more smaller messages when the number of blocks $q$ increases.

The matrix multiplication algorithm that was used in [11] for benchmarking differs from this algorithm as it generates a different communication pattern and is not memory efficient. Also, their implementation uses an $IJK$ loop for doing the local products, which leads to different behavior due to caching effects. When using an $IJK$ loop with the algorithm described above, the performance increases for small block sizes on low latency interconnection networks, because the blocks can then fit into the processor cache. When using BLAS, this behavior only occurs for smaller block sizes, which is more desirable for our purposes because it makes it easier to predict local computation times. Also, the BLAS implementation achieves better overall performance, as can be seen from Fig. 1.

### 6. Matrix Multiplication Experiments

The experiments were conducted on the same systems as the benchmarks. For each system and library, the running time for matrix sizes between $100 \times 100$ and $2000 \times 2000$ was measured. The number of blocks $V_{IJK}$ in the cube was set to the values $3^3, 4^3, 5^3, \ldots, 9^3$, leaving out the $3^3, 4^3$ values for larger numbers of processors when they would lead to underpartitioning (i.e. $q^{2/3} < p$).

Very small messages can arise due to the alignment of stored and transmitted blocks, which do not have the same width when $q^{2/3} > p$. Spikes in the running time can occur when $q^{1/3}$ is a perfect multiple of $\sqrt{p}$. For a similar reason there may be spikes when the matrix size $n$ is not a perfect multiple of $q^{1/3}$. This makes it harder to predict the performance exactly for different values of $q$ with the same value of $g$. In particular, the performance will decrease for larger values of $q$, because there is presumably a higher effective value of $g$ for the smaller blocks. For our prediction, the value of $q$ for the largest matrix size from Tbl. 3 is used. The value of $l$ was assumed to be the value measured for all-to-all communication.

Figures 4 and 6 show the resulting running times for the different values of $q$ as a function of matrix size, averaged over 5 runs each. The results for 10 and processors on aracari and PUB/MPI were left out to conserve space. On 10 processors, a similar behavior as on 16 processors can be seen, on 32 processors the predictions only fit approximately for Oxtool. PUB/MPI show a similar behavior to Oxtool on argus, although the prediction matches best for MPI. The dashed lines show the predicted values according to the parameters given on top of each plot. The results for some values of $q$ were left out to conserve space, they show similar characteristics as the other ones.

The predictions match the experimental results well for small numbers of processors. As discussed above, the value of $g$ that is used is not necessarily equal
Fig. 4. Matrix multiplication — overall running times on aracari
to the practical bandwidth when the block size decreases. The predicted spikes for block alignment mismatches occurred for PUB on 4 processors; for the other libraries, similar effects could only be observed when using more than 32 processors (on aracari). This can be explained with the uneven communication characteristics that PUB showed in the experiments from Section 4. For 16 and 32 processors the predictions are less accurate, because the communication pattern of the algorithm becomes less similar to the all-to-all exchange used for benchmarking. This can also explain the large difference between the prediction and running time for 32 processors on aracari/MPI. For a higher number of processors, the size of the blocks stored on each processor becomes smaller, leading to smaller messages.

The contribution of the latency \( l \) to the running time of the algorithm can be considered negligible in case of 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 microseconds, the synchronization time only adds up to 0.37 seconds. In general, \( l \) only becomes significant when the communication network has high latency and the number of supersteps is large. The performance model becomes increasingly more sensitive to inaccuracies in the parameter \( g \) when the communication volume per superstep is low, because then the practical values of \( g \) vary more. This happens in particular
for larger values of $p$ and $q$, where there are more smaller matrix blocks that have to be transferred. For larger $n$ the communication volume increases, which leads to more stable practical values of $g$.

To compare the performance and scalability of the libraries, Figures 5–6 show comparison graphs for the speedup (running time of the algorithm on one node divided by its running time on $p$ nodes with parameter $q$) on the different machines.

The overall performance of the Oxford BSP toolset was better than that of PUB; the latter showed greater sensitivity to different conditions on a busy communications network. Only for small problem sizes on the slower communications network was PUB able to slightly outperform Oxford BSP toolset. PUB also showed less stability in its performance even with the fast communications network, and communication times increased once a certain message count and size was reached. This also showed in the experiments from [11]. MPI occasionally showed spikes of very long communication time. These spikes occurred randomly and were probably due to different traffic on the communications network.

All implementations except for PUB achieved good speedup on aracari. PUB was only able to deliver good speedup when the communication volume was small (in particular the number of messages), otherwise the performance decreased. Best scalability was achieved by the MPI implementation, except on 10 processors where Oxtool showed slightly better performance. For higher values of $q$ and small matrix sizes superlinear speedup occurs when the blocks for the local products fit into the CPU cache and thus reduce local computation times. When using the algorithm for practical purposes, $q$ can thus be chosen according to the ratio of $g$, $l$ and the matrix size for which an optimum processing flop rate is achieved.

On argus none of the libraries achieved good speedup, this is due to the fact that the individual nodes are very fast and have a slow communications network. The communication time thus becomes the predominant part of the overall running time, another factor is also the overhead created by running on top of MPI. For the given matrix sizes, individual nodes were able to perform the multiplications faster than the parallel versions. However, PUB and Oxtool showed an advantage over plain MPI, for larger messages and higher communication volume the performance remained more stable there.

7. Results and Conclusions

Experiments have been conducted for different simple one superstep communication patterns (all-to-all and local shifts) and dense matrix-matrix multiplication using BSPlib style communication libraries. The Oxford BSP toolset, PUB (both compiled on top of MPI) and a ‘naive’ MPI implementation were compared on two different message passing platforms. The running times were compared to the predicted ones according to the BSP model. Furthermore, the relative performance and scalability of the implementations were compared.

Altogether the experiments have shown that the performance of a BSP algorithm’s implementation depends on how suitable the optimizations of the BSP library implementation are for the parallel computer it is run on, and that for fixed problem sizes and an adequate choice of BSP parameters, realistic bounds for the running time can be found. However, it can prove difficult to find these parameters
when the algorithm produces different communication patterns from the benchmarking measurements.

The BSP model with fixed values of $g$ and $l$ alone is not sufficient for predicting the performance exactly, in general only an upper or lower bound can be obtained. The performance of a communication primitive obviously depends on message size and count per superstep, which gives the graph of a curved surface like in Fig. 3 instead of just a plane for a constant value of $g$. Also the graphs for different communication primitives (get, put, send) can differ. Good estimations are only possible, when the algorithm’s communication pattern does not involve communication patterns where varying message sizes lead to strong variations in the value of $g$. An improvement could possibly be made by changing the communication cost formula to include a size dependent value for $g$. Also, Bianco and Pucci provide a study of ‘BSP-like cost functions’ in [1].

8. Further Research

Further experiments are currently being run for higher numbers of processors and bigger matrices. Since a shared memory machine has recently become available at the Centre for Scientific Computing, it seems reasonable to port the libraries under consideration there and run the same benchmarks for comparing the performance. Also, the remaining communication library SSCRAP can be integrated. Further evaluation will be done by implementing algorithms that generate a different communication pattern. Another possibility would be to consider different cost models that are more suitable for SMP architectures (e.g. the D-BSP model, see [2]), or further explore the approach of using different BSP-like cost functions [1].

9. Acknowledgements

I would like to thank my supervisor Alexander Tiskin for his support and guidance during my work, as well as the referees for their suggestions and comments.

References


CGMLib: [http://www.scs.carleton.ca/~cgm/](http://www.scs.carleton.ca/~cgm/)


PUB library: [http://www.uni-paderborn.de/~bsp/](http://www.uni-paderborn.de/~bsp/)
