Experimental Evaluation of BSP Programming Libraries

Peter Krusche

Department of Computer Science
Centre for Scientific Computing
University of Warwick
peter@dcs.warwick.ac.uk
Outline

Motivation...
Study and compare the communication characteristics and performance predictability of ‘BSP-style’ communication libraries.

Outline
1. The BSP Model
2. BSP Programming Libraries
3. Benchmarking
4. Performance and Predictability
The BSP Model

The BSP model for parallel algorithms was used with some slight adaptations.

The model has been adapted here to use seconds instead of flops as a base unit for the running time:

- $p$ identical processor/memory pairs (computing nodes), computation speed $f$
- Arbitrary interconnection network, latency $l$, bandwidth $g$
The BSP Model

- Programs are SPMD
- Execution takes place in *supersteps*
- Cost Formula: \( T = f \cdot W + g \cdot H + l \cdot S \)
- As a base unit for communications, 8-byte doubles will be used
BSP Programming

‘BSP-style’ programming using a conventional communications library (MPI/Cray shmem/...)
- Barrier synchronizations for creating superstep structure
- Many libraries already provide functionality for one sided communication/direct remote memory access (DRMA)

Using a specialized library (The Oxford BSP Toolset/PUB/CGMlib/...)
- Specialized communication primitives (bulk synchronous message passing/DRMA)
- Some libraries (Oxford Toolset, PUB) include optimized barrier synchronization functions and routing
- Higher level of abstraction
The BSPlib Standard

Communication primitives:

- **DRMA**: buffered and unbuffered put, get
- **BSMP**: send and move
- Synchronization
- Combining and Broadcasting

For the experiments, a BSPlib-style wrapper library was created.
BSPlib Implementations

The Oxford BSP Toolset
- Supports 3 kinds of base architecture: message passing, shared memory, DRMA
- Experiments used message passing MPI interface
- Last release from ’98, compatibility issues on more modern systems

PUB
- Support for message passing and shared memory architectures
- Experiments used message passing MPI interface
- Additional support for oblivious synchronization, processor groups
- Less trouble with setup on all systems
- Advanced functionality e.g. for process migration
Other Libraries

CGMlib
- Runs on top of message passing MPI
- Includes set of algorithms for sorting, list ranking, etc.
- Abstract C++ interface
- Lists of abstract datatypes with constant size are used for data exchange

SSCRAP
- Uses MPI (message passing) or Posix (SHMEM) for data exchange
- Support for DRMA, BSMP, conventional message passing, collective operations, etc.
- ’Soft’ synchronization (send or receive)
- C++ interface
‘BSP-style’ Programming in MPI

Approach here: a BSPlib style MPI-1 library was implemented naively (without message combining, etc.)

- Isend/Recv for data exchange
- Barrier synchronization
- Emulated DRMA on top

Advantage: no overhead for `send/put`

Drawback: high latency, presumably overhead for `get` operations
Systems used

Measurements on parallel machines at the Centre for Scientific Computing:

**aracari**: IBM cluster, 64 × 2-way SMP Pentium3 1.4 GHz/128 GB of memory
(Interconnection Network: Myrinet 2000, MPI: *mpich-gm*)

**argus**: Linux cluster, 31 × 2-way SMP Pentium4 Xeon 2.6 GHz processors/62 GB of memory
(Interconnection Network: 100Mbit Ethernet, MPI: *mpich-p4*)
Measuring $f$

Measuring algorithm performance on one node:  

Measuring computation time separately in one run:

(Example for Matrix-Matrix multiplication)
Measuring $g$ and $l$

Problems encountered: realistic values of $g$ and $l$ depend on

- The number of processors that are used
- The communications pattern
- The communication volume

E.g. for all-to-all communication on aracari
Bandwidth Surface \((\text{aracari})\)

For a better picture, the effective bandwidth can be sampled depending on message size and count.

All-to-all communication on \text{aracari}:
Bandwidth Surface (aracari)

For a better picture, the effective bandwidth can be sampled depending on message size and count.

All-to-all communication on aracari:

![Graph showing time per element in microseconds (us) vs message size and number of messages for PUB 16 nodes and OXTOOL 16 nodes.]
Bandwidth Surface (aracari)

For a better picture, the effective bandwidth can be sampled depending on message size and count.

Random permutation on aracari:
Bandwidth Surface (*aracari*)

For a better picture, the effective bandwidth can be sampled depending on message size and count.

Random permutation on *aracari*:

![Graph showing Bandwidth Surface for PUB 16 nodes and OXTOOL 16 nodes.](image-url)
Bandwidth Surface (argus)

The picture looks different on the slower communications network

All-to-all communication on argus:

![Graph showing bandwidth surface performance]
Bandwidth Surface (argus)

The picture looks different on the slower communications network

All-to-all communication on argus:

![Bandwidth Surface Graph]

- Bandwidth Surface (argus)
- The picture looks different on the slower communications network
- All-to-all communication on argus:

![Graph showing time per element vs. message size and number of messages for PUB and OXTOOL on 4 nodes]
Bandwidth Surface (argus)

The picture looks different on the slower communications network

Random permutation on argus:

![Bandwidth Surface Graph](image-url)
Bandwidth Surface (argus)

The picture looks different on the slower communications network

Random permutation on argus:

![3D graph showing time per element vs. message size and number of messages for PUB 4 nodes and OXTOOL 4 nodes.](image)
## Latency

The latency can be measured for synchronizations preceded by different types of communication:

<table>
<thead>
<tr>
<th>Type</th>
<th>MPI</th>
<th>Oxtool</th>
<th>PUB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>aracari</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4 processors)</td>
<td>210 $\mu$s</td>
<td>43 $\mu$s</td>
<td>39 $\mu$s</td>
</tr>
<tr>
<td>(low)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(high)</td>
<td>230 $\mu$s</td>
<td>67 $\mu$s</td>
<td>55 $\mu$s</td>
</tr>
<tr>
<td>(all-to-all)</td>
<td>252 $\mu$s</td>
<td>89 $\mu$s</td>
<td>72 $\mu$s</td>
</tr>
<tr>
<td><strong>aracari</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(32 processors)</td>
<td>2203 $\mu$s</td>
<td>621 $\mu$s</td>
<td>142 $\mu$s</td>
</tr>
<tr>
<td>(low)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(high)</td>
<td>2242 $\mu$s</td>
<td>638 $\mu$s</td>
<td>163 $\mu$s</td>
</tr>
<tr>
<td>(all-to-all)</td>
<td>2881 $\mu$s</td>
<td>1250 $\mu$s</td>
<td>750 $\mu$s</td>
</tr>
<tr>
<td><strong>argus</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4 processors)</td>
<td>5642 $\mu$s</td>
<td>796 $\mu$s</td>
<td>975 $\mu$s</td>
</tr>
<tr>
<td>(low)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(high)</td>
<td>5789 $\mu$s</td>
<td>1442 $\mu$s</td>
<td>1176 $\mu$s</td>
</tr>
<tr>
<td>(all-to-all)</td>
<td>5086 $\mu$s</td>
<td>1613 $\mu$s</td>
<td>871 $\mu$s</td>
</tr>
</tbody>
</table>
Benchmark Summary

Bandwidth depends on message count, message size and the communications pattern

On aracari:
- Best all-to-all performance: Oxtool
- Best random permutation performance (few messages): PUB, \( > 64 \) messages: Oxtool
- Best self communication performance (few messages): PUB, \( > 32 \) messages: Oxtool
- MPI: good performance when message size is large
Benchmark Summary

Bandwidth depends on message count, message size and the communications pattern

On argus:
- Best all-to-all performance: PUB
- Random permutation: little difference between PUB and Oxtool
- Best self communication performance (few messages): Oxtool
- MPI: good performance when message size and count are larger than 16/32 doubles
Benchmark Summary

Latency:

- PUB consistently has best latency (without using the faster ‘oblivious’ synchronization)
- As expected, ‘naive’ MPI library has highest latency
BSP Matrix-Matrix Multiplication

We want to compute the product of two dense $n \times n$ matrices $A$ and $B$

Simple formula:

$$c_{ik} = \sum_{j=1}^{n} a_{ij}b_{jk}, \text{ having } A = [a_{ij}], \quad B = [b_{ij}], \quad C = [c_{ij}]$$
BSP Matrix-Matrix Multiplication (2)

Block decomposition into \( q \) blocks for memory efficient parallel algorithm:

\[
C_{IK} = \sum_{J=1}^{q^{1/3}} V_{IJK} \quad \text{with} \quad I, K = 1, 2, \ldots, q^{1/3}
\]
Why this algorithm?

- Communication block size can be controlled by parameter $q$
- Message combining has to be used when using fixed initial data distribution (block-cyclic with block width $n / \sqrt{p}$)
- Can be compared e.g. to PBLAS
- ‘Nice’ version can predistribute the blocks before the computation to avoid spikes because of data distribution
Prediction model

BSP running time:

\[ T = f \cdot \left\lceil \frac{q}{p} \right\rceil \cdot \frac{n^3}{q} + \]

\[ g \cdot \left\lceil \frac{q}{p} \right\rceil \cdot \frac{n^2}{q^{2/3}} \cdot \left( 2 + \frac{1}{q^{1/3}} \right) + \]

\[ l \cdot 2 \left\lceil \frac{q}{p} \right\rceil \]

Two matrices are transferred row by row

→ value of \( g \) is taken from the red line as value for maximum matrix size
Prediction results on aracari

Oxtool, using 4 processors

\[ p = 4, \quad 1/f = 4 \times 10^8 \quad g = 3.5 \times 10^{-7} \quad l = 8.9 \times 10^{-5} \]

\[ p = 4, \quad 1/f = 4 \times 10^8 \quad g = 3.5 \times 10^{-7} \quad l = 8.9 \times 10^{-5} \]

\[ q = 27 \quad \text{mean error 9.89\%} \]

\[ q = 125 \quad \text{mean error 10.60\%} \]

\[ q = 343 \quad \text{mean error 13.06\%} \]

\[ q = 729 \quad \text{mean error 13.56\%} \]

- prediction \( q = 27 \)
- prediction \( q = 125 \)
- prediction \( q = 343 \)
- prediction \( q = 729 \)
Prediction results on aracari

PUB, using 4 processors

\[ p = 4, \ \frac{1}{f} = 4 \times 10^8 \ g = 2.5 \times 10^{-6}, \ l = 7.2 \times 10^{-5} \]

- Prediction results on aracari
- Prediction results, more processors
- Speedup Results (1)
- Speedup Results (2)
- Performance Comparison with PBLAS

\[ q=27 \text{ mean error 52.68}\% \]
\[ q=125 \text{ mean error 91.92}\% \]
\[ q=343 \text{ mean error 110.39}\% \]
\[ q=729 \text{ mean error 120.03}\% \]

- prediction \( q=27 \)
- prediction \( q=125 \)
- prediction \( q=343 \)
- prediction \( q=729 \)
Prediction results on aracari

MPI, using 4 processors

\[ p = 4, \ 1/f = 4e+08 \ g = 3.4e-07, \ l = 0.00025 \]

\[ \text{Running time [s]} \]

\[ \text{Matrix Size} \]

- Prediction q=27 mean error 4.04%
- Prediction q=125 mean error 4.92%
- Prediction q=343 mean error 7.44%
- Prediction q=729 mean error 8.20%
Prediction results, more processors

Oxtool, using 32 processors

\[
p = 32, \quad 1/f = 4e+08, \quad g = 5.3e-07, \quad l = 0.0013
\]

- Prediction results, more processors
- Oxtool, using 32 processors
  - q=343 mean error 20.36%
  - q=512 mean error 31.71%
  - q=729 mean error 34.20%
  - prediction q=343
  - prediction q=512
  - prediction q=729
Speedup Results (1)

aracari, using 16 processors
Speedup Results (2)

aracari, using 32 processors (spikes when matrix size $\text{mod } 6 == 0$)

![Speedup Results (2) Diagram]
Performance Comparison with PBLAS

![Graph comparing performance of different libraries with varying matrix sizes.](image)

- PBLAS, 36 nodes
- PBLAS, 16 nodes
- MPI, 36 nodes
- MPI, 16 nodes
- Oxtool, 36 nodes
- Oxtool, 16 nodes
- PUB, 36 nodes
- PUB, 16 nodes

- Performance Comparison with PBLAS

Conclusion
Performance Comparison with PBLAS

![Graph comparing performance of different libraries against matrix size. The x-axis represents matrix size in thousands, ranging from 2000 to 10000. The y-axis represents flop/s, ranging from 2G to 12G. Four lines represent PBLAS, PUB, MPI, and Oxtool.]
Summary

- Despite restrictions due to BSP model, all implementations reach good speedup on aracari when number of blocks is low
- Overall benchmark results look better for PUB
- Oxtool has best matrix multiplication performance on aracari (Myrinet)
- PUB has best matrix multiplication performance on argus (Ethernet)
- Predictably no real speedup on argus, due to slow communications network and fast nodes
- Performance of simple BSP algorithm is comparable with PBLAS
Further Work

- Run experiments on shared memory machine
- Use more different communication patterns for benchmarking
- Study other algorithms with different communication patterns
- Keeping simplicity, extend prediction model for more accuracy