Implementing BSPonMPI 0.1
Lessons Learned & Results

Wijnand Suijlen

June 29, 2006

Abstract

BSPonMPI implements the BSPlib standard and runs on all machines which have MPI. During the development a few lessons were learned about writing fast C programs. Additionaly the library is benchmarked and results show that it can compete with Oxford BSP Toolset. Remarkable is the observation that the MPI implementation of the Oxford BSP Toolset scales better then the native implementation.

1 Introduction

BSPonMPI is a platform independent software library for developing parallel programs. It implements the BSPlib standard [5] and runs on all machines which have MPI [2]. This last property is the main feature of this library and with this feature it distinguishes itself from the two major BSP libraries: Oxford BSP Toolset [4] and PUB [1]. Both are implemented for specific hardware platforms (e.g. Cray T3E or SGI Origin) and they have a platform independent version on top of MPI. However the architecture of their software library is optimised for the use of hardware specific features. Building on top of MPI was never their primary objective. The mission of BSPonMPI is to be the fastest BSP library which uses MPI.

During the implementation of this library and the quest for better performance I learned a few things. After finishing the implementation they seem obvious, but they were not that obvious beforehand. This report is about the experience gained while writing this library and an evaluation of the performance. Code documentation, design issues and library usage can be found on the project homepage [7].

Throughout this document I use some terminology which is common in the field of BSP programming. Here is short glossary

g Throughput parameter. It stands for the costs of sending one extra data element (e.g. measured in seconds per byte).

l Latency parameter. It denotes the minimum costs of communication or, more specific, the cost of one `bsp_sync()` in which 0 bytes are communicated (e.g. measured in seconds).

h-relation Communication pattern. h stands for the amount of data sent by each processor. If this differs among processors, it denotes the maximum. An h-relation is full, if all processors send at least one data element.

2 Lessons learned

Before this project my only experience in High Performance Computing (HPC) was a computer exercise about computing the homogeneous Poisson equation on a rectangular grid using Fortran. In this exercise the data structure was obvious and very simple. Additionally Fortran does not allow many complex data structures, because it is not aware of pointers. In C anything is possible and therefore it may be tempting to use the full range of C's pointer abilities.

In fact: I was tempted. Especially because I am used to an object oriented programming style. If I needed to store a set of variables of which I thought they belonged to each other, I defined a `struct` for it and gave it an address: statically or dynamically. So the first time implementing the communication buffer I constructed it as a linked list, where memory is allocated dynamically for each communication instruction object. This solution proved to be very slow for three different reasons:

1. In order to send the communication buffer using `MPI_Alltoallv`, it still had to be copied to an array.
2. The communication data could be scattered throughout the heap[1] may result in a lot of cache misses.
3. calls to malloc and free are not very cheap.

The greatest increase of performance was gained by doing the memory management myself: allocate an array at the start of the program and resize it if necessary. On Teras using 2 processors (see 3.1) the BSP throughput parameter g was reduced by 50% (see figure 1) as memory expansions now only happen a few times during the program instead of at every addition of a communication instruction.

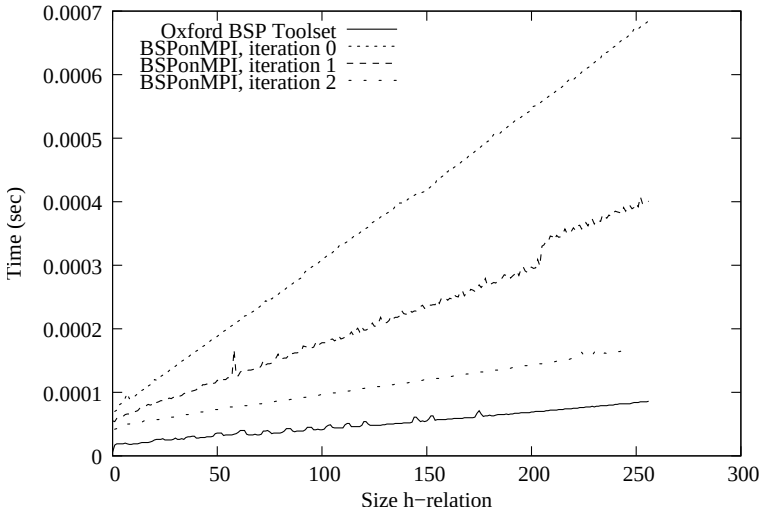

Figure 1: Compares Oxford BSP Toolset to three versions of BSPonMPI: **iteration 0** is an implementation having a linked list as communication buffer; **iteration 1** is the first implementation having arrays as communication buffer; **iteration 2** is the current version in which some other optimisations have been done

Two lessons are learned here:

1. *Keep it simple*: Implement your algorithms as you would do in a basic programming language like Fortran or Basic. In this case arrays and array operations are used instead of linked list and complex pointer operations.
2. *Put all complex / slow code in exceptions which happen seldom*: In this case only a few times memory was expanded instead of at each action.

This does not only result in faster code, it also provides the compiler with more opportunity for optimisation and makes the code more readable. Applying these lessons on other parts of the code reduced g again by 50 % (see figure 1).

If you follow these rules, you almost automatically obey the standard optimisations rules:

- When optimising code, focus on the lines which are executed most frequently, e.g.: optimise the body of a loop.

[1] In general a program has two memory regions in which it stores its data: the stack and the heap. The stack contains function parameters, return values, variables and arrays of fixed size. The heap contains all dynamically allocated data.

- Limit the use of branch statements, e.g. `if` or `switch`
- Limit calls to `extern` functions
- Access data using aligned addresses.
- Use simple language, i.e. use

  ```
  for (i = 0; i < 10; i ++)
    sum += a[i];
  ```

  in stead of

  ```
  b = a + 9;
  while (b >= a)
    sum += *b;
  ```

- Supply the compiler with hints, i.e. use `const`, `inline` and `restrict`.

3 Results

3.1 Test environment

The performance of BSPonMPI 0.1 has been measured on two different machines:

- Teras, which is a SGI Origin 3800. On this machine a version of Oxford BSP Toolset v1.4 is installed which is specially built for this hardware platform.
- Aster, which is a SGI Altix 3700. This machine has a version of Oxford BSP Toolset v1.4 built on top of MPI.

The benchmark program used is a modified version of 'bench' which is part of 'BSPedupack' [3]. The modified version can be found at the BSPonMPI homepage [7]. Whereas the original version only measured the speed of a *put* operations, the modified version can also measure using *get* and *send* operations. Output of 'bench' is provided in figures 2–7 on the following pages.

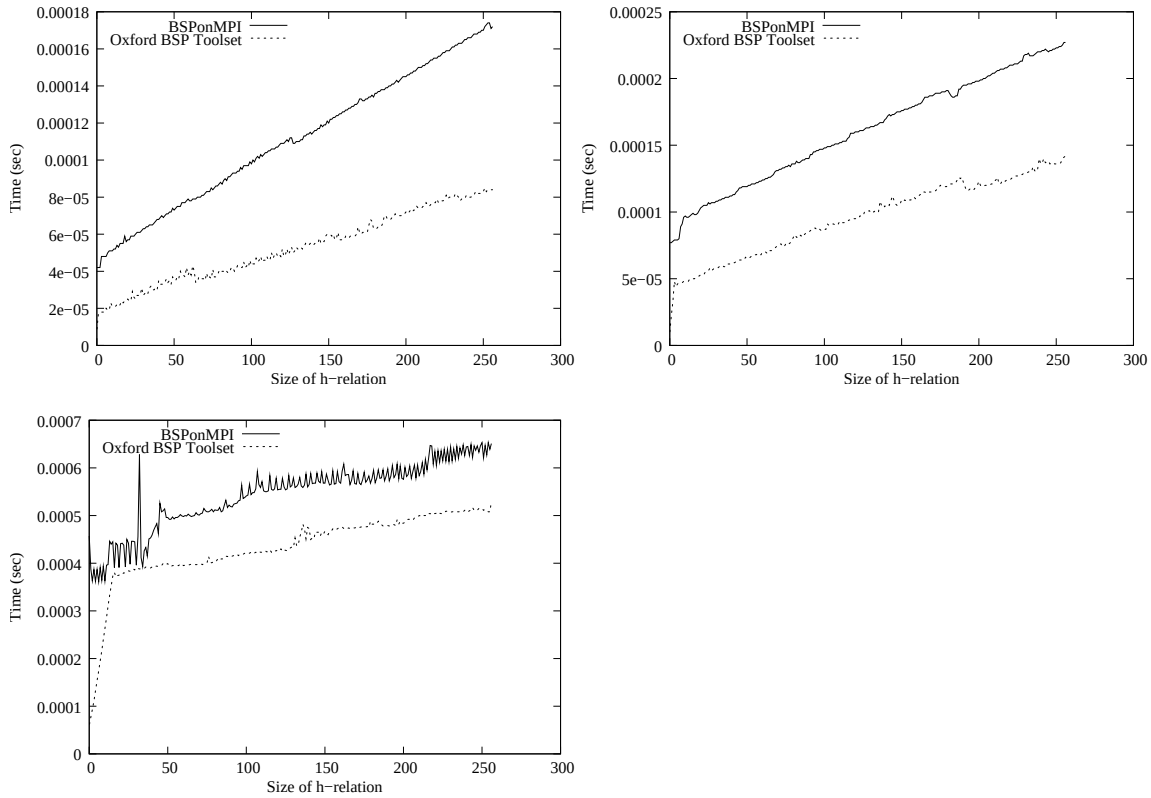

Figure 2: Comparison of BSPonMPI and Oxford BSP Toolset using *put* operations on 2, 4 and 16 processors of the computer teras (from left to right and top to bottom).

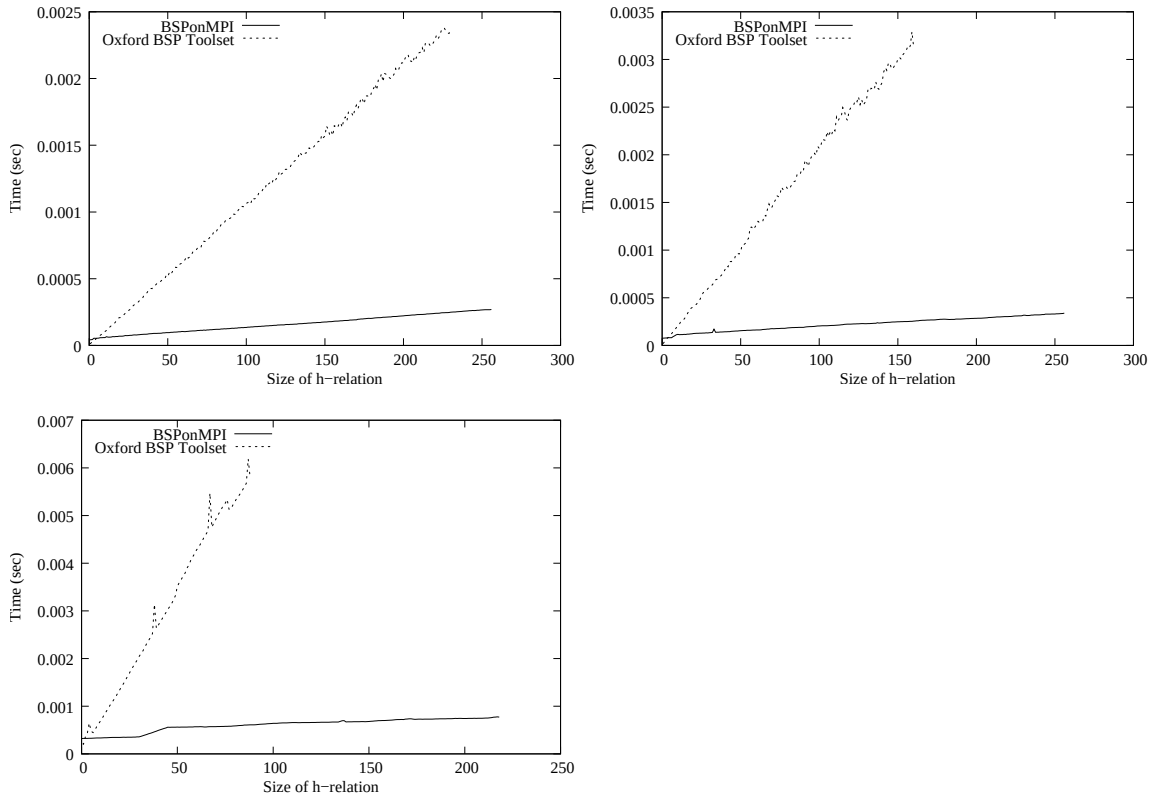

Figure 3: Comparison of BSPonMPI and Oxford BSP Toolset using *get* operations on 2, 4 and 16 processors of the computer teras (from left to right and top to bottom).

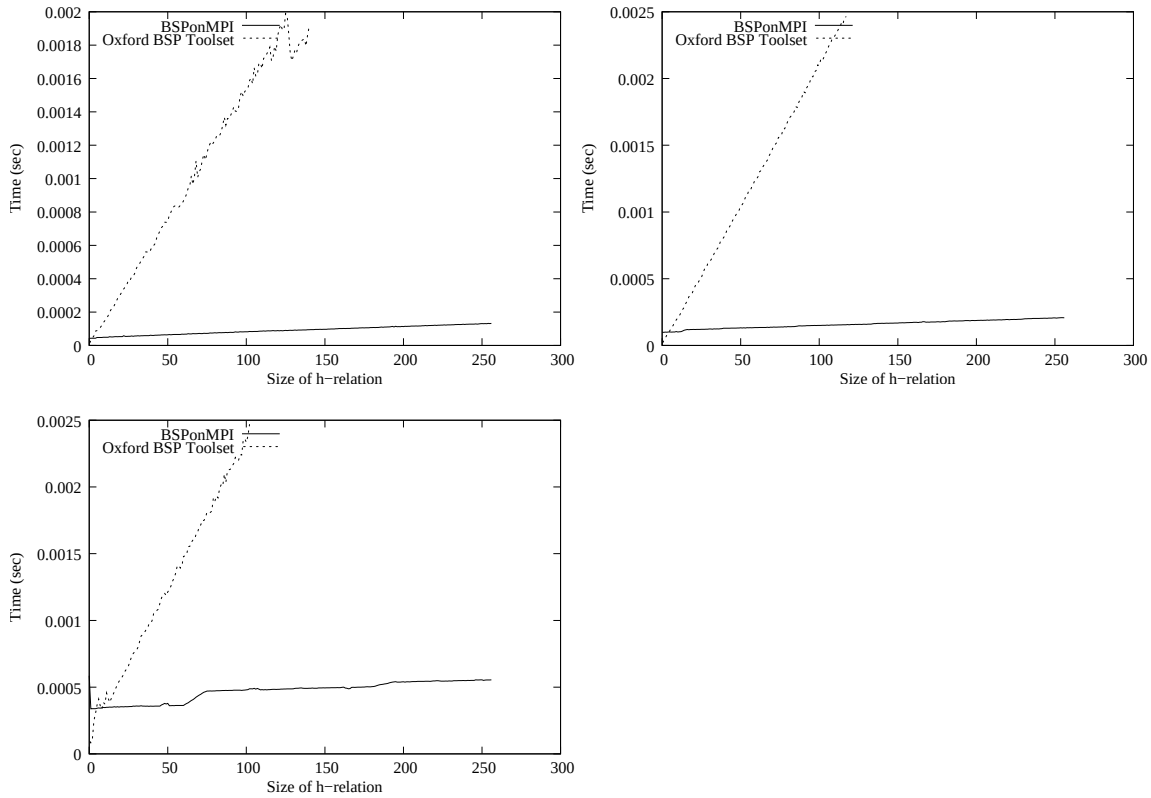

Figure 4: Comparison of BSPonMPI and Oxford BSP Toolset using *send* operations on 2, 4 and 16 processors of the computer teras (from left to right and top to bottom).

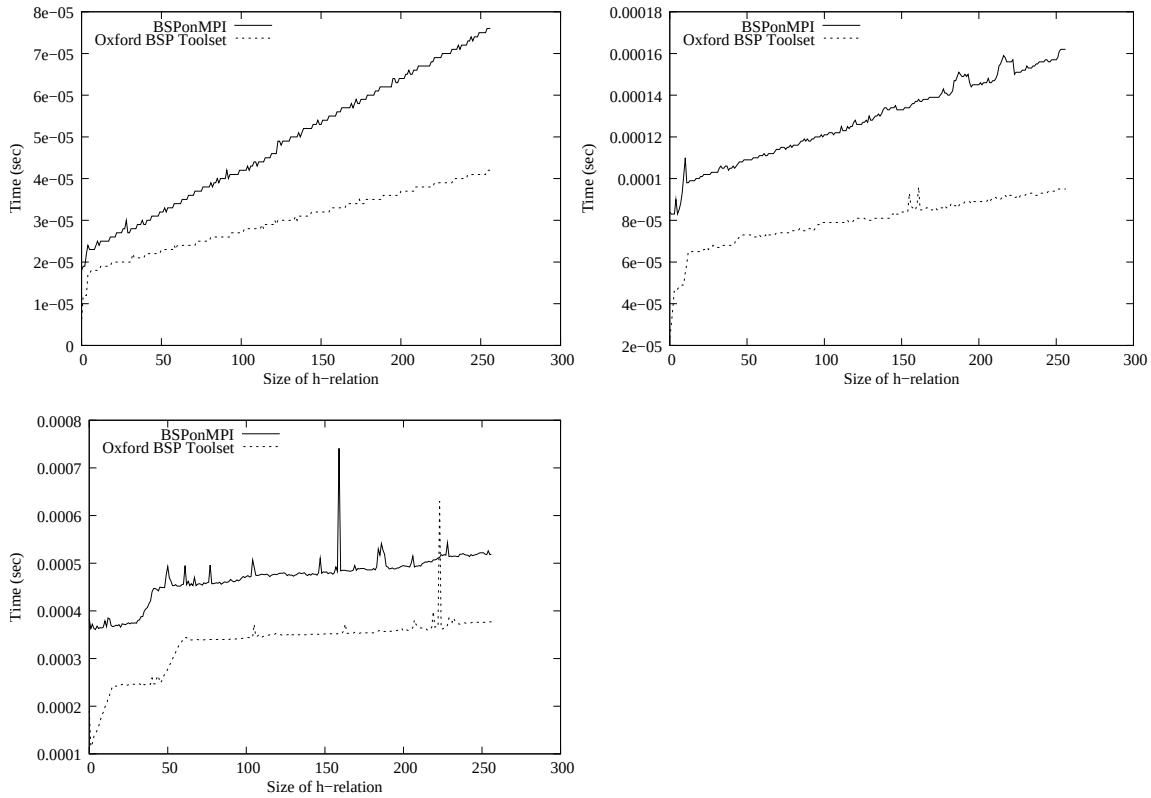

Figure 5: Comparison of BSPonMPI and Oxford BSP Toolset using *put* operations on 2, 4 and 16 processors of the computer aster (from left to right and top to bottom).

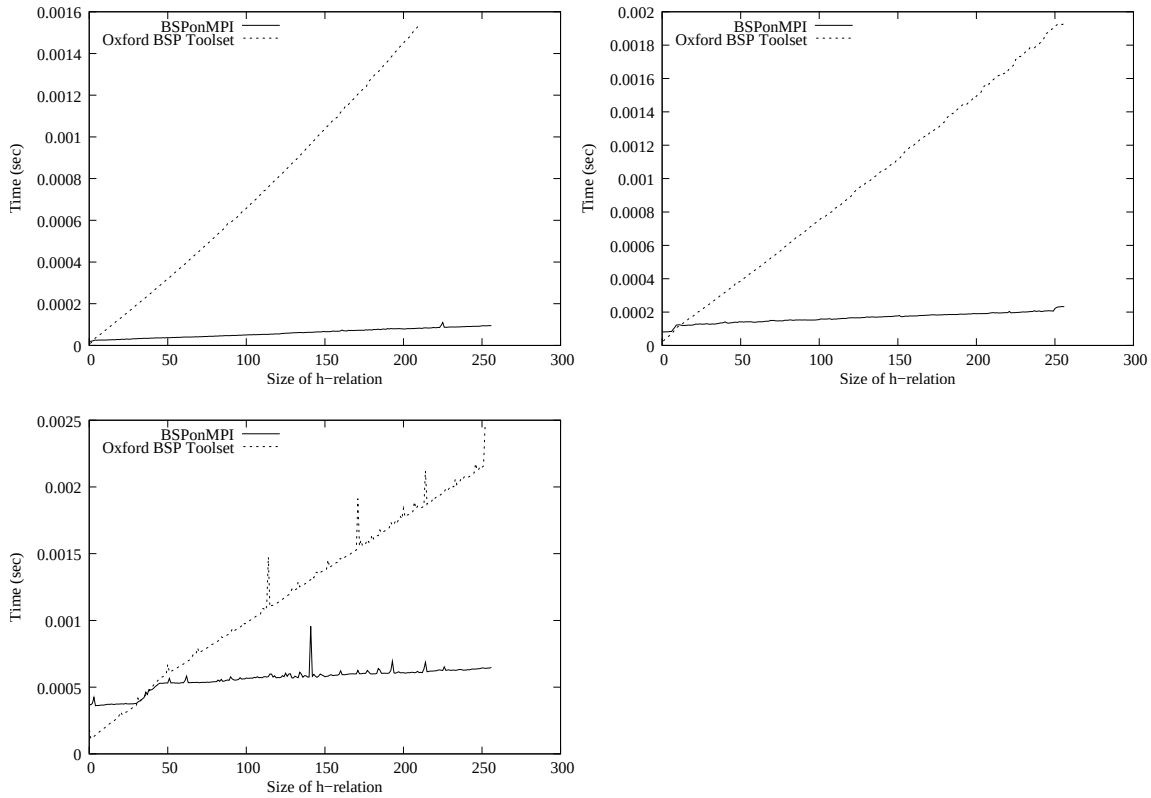

Figure 6: Comparison of BSPonMPI and Oxford BSP Toolset using *get* operations on 2, 4 and 16 processors of the computer aster (from left to right and top to bottom).

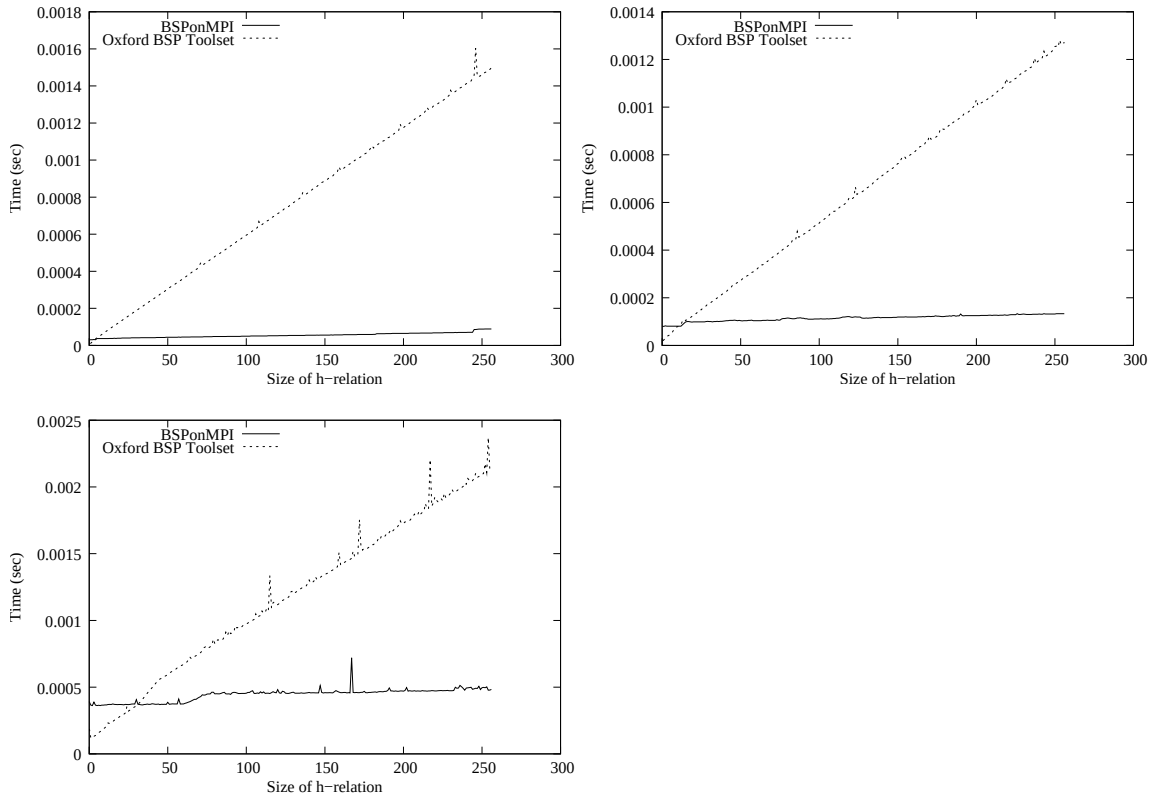

Figure 7: Comparison of BSPonMPI and Oxford BSP Toolset using *send* operations on 2, 4 and 16 processors of the computer aster (from left to right and top to bottom).

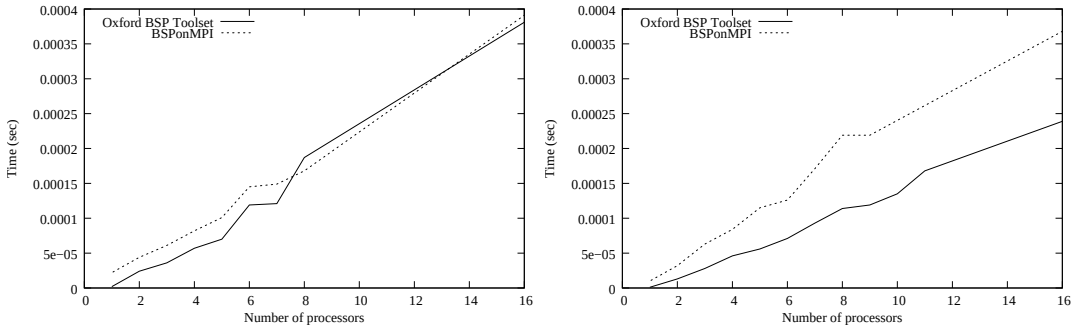

Figure 8: This figure shows the cost of a full h-relation while scaling up the number of processors. On the left results on Teras are shown, on the right Aster

The purpose of BSPonMPI is to beat Oxford BSP Toolset on Aster as it was supposed to be suboptimal. BSPonMPI does not not succeed in that when considering *put*s only, but astonishingly it beats Oxford on both machines when measuring using *get*s and *send*s. BSPonMPI was designed to process *put*s, *get*s and *send*s in an equivalent manner. The benchmark results confirm this as experiments on equally sized h-relations show equal timings. Apparently Oxford BSP Toolset processes them quite differently.

You may also notice that performance differences becomes less apparent when the number of processors increases. From figures 2–7 it can be seen that latency increases very fast, while the throughput parameter g shows a more moderate increase. Obviously the latency becomes more and more dominant when the number of processors is scaled up. In figure 8 the latency is shown for a varying number of processors. It seems that BSPonMPI and its competitor perform equally on Teras, whereas BSPonMPI performs always worse on Aster. Again something very strange happened here.

4 Conclusion

The pros and cons of BSPonMPI are

- *put* operations are not yet very quick
- high latency on non-full h-relations.
- it is not yet capable of detecting BSP programming errors. Up to now all effort has been directed in implementing a fast library; not a robust one
+ *get* and *send* operations are very fast.
+ The source code is platform independent.
+ It transforms any BSP program into an MPI program. All MPI tools, such as profilers, will work with them.

Although this implementation is not yet fully optimised, we see that BSPonMPI can already compete with the native and MPI versions of Oxford BSP Toolset. Which library is best, depends on the application. Remarkable is the fact that BSPonMPI performs best when compared to the native version of Oxford BSP Toolset. As BSPonMPI is the constant factor in this comparison, one may wonder whether the MPI version of Oxford BSP Toolset performs better than the native version.

I expect to gain more performance in a subsequent version of BSPonMPI, because:

latency l can be reduced by 25% merging two `MPI_Alltoall()`'s which communicate buffer sizes. More may be gained by determining whether it is really necessary to communicate buffer sizes.

throughput g can be reduced by 30% by changing the communication buffer.

This may yield a library which scales much better and beats the native and MPI versions of Oxford BSP Toolset.

References

[1] Paderborn University BSP-library
`http://wwwcs.uni-paderborn.de/~bsp`

[2] MPI 1.1 Standard
`http://www.mpi-forum.org/docs/docs.html`

[3] Rob Bisseling. BSPedupack v1.0
`http://www.math.uu.nl/people/bisselin/software.html`

[4] Jonathan Hill. Oxford BSP Toolset v1.4
`http://www.bsp-worldwide.org/implmnts/oxtool`

[5] Jonathan Hill, Bill McColl, Dan Stefanescu, Mark Goudreau, Kevin Lang, Satish Rao, Torsten Suel, Thanasis, Tsantilas and Rob Bisseling. BSPlib: The BSP programming library. *Parallel Computing,* 24:1947–1980, 1998.

[6] Jonathan M. D. Hill, David B. Skillicorn. Lessons Learned from Implementing BSP. *HPCN Europe 1997,* 762–771

[7] Wijnand Suijlen. BSPonMPI homepage
`http://bsponmpi.sourceforge.net`