Utrecht University

BACHELOR THESIS

A C++ Object Oriented library for the BSP model: reducing the cost of communication and synchronization

---

Author:
Mick VAN DUIJN

Supervisor:
Prof. dr. Rob H. BISSELING

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science, Mathematics

in the

Faculty of Science,
Mathematical Institute

June 16, 2016

UTRECHT UNIVERSITY

Abstract

Faculty of Science,
Mathematical Institute

Bachelor of Science, Mathematics

A C++ Object Oriented library for the BSP model: reducing the cost of communication and synchronization

by Mick VAN DUIJN

At the beginning of the work for this thesis, the aim is to devise a comparable baseline for the benchmarking of MulticoreBSP and Zefiros-BSPLib for computations, and to reduce the communication cost parameter g and the synchronization cost parameter l. After that, the aim is to extend the library to more extensive cases, such as specialized synchronizations aimed at a specific subset of communication patterns. Finally, some existing programs are compiled with the Zefiros-BSPLib implementation as well as the MulticoreBSP implementation on the same machine, for a good comparison of real-world examples. This thesis is focused on a **shared memory** implementation. Many of the ideas of this shared memory implementation can be ported to distributed memory implementations, but this is out of the scope of this thesis.

Acknowledgements

First of all, I would like to thank my supervisor Rob Bisseling for the inspiring course on Parallel Algorithms and for allowing me to write this thesis with his supervision. He has helped me greatly in shaping the content of this thesis, and keeping me focused on what is important for this thesis. His book [1] helped in understanding the way the algorithms in this thesis were parallelized. The algorithms and code presented in this thesis are from this book, and are not meant to be innovative, but rather they are used as realistic test cases for the improvements made in this thesis.

Secondly, I would like to thank Paul Visscher for the implementation we wrote together, Zefiros-BSPLib. This has been the basis of chapter 2. We worked together on implementing the rather extensive basis for the Zefiros-BSPLib implementation. He also helped me brainstorm on some of the improvements that could be made for this thesis.

Contents

Abstract ii

Acknowledgements iii

Preface 1

1 The BSP model 2
- 1.1 The model 2
- 1.2 Supersteps 2
- 1.3 Different types of communication 3
  - 1.3.1 Put 3
  - 1.3.2 Get 3
  - 1.3.3 Send 4
  - 1.3.4 Registration of variables 4
- 1.4 Synchronization 5
- 1.5 The BSP cost 5
- 1.6 Example BSP algorithm 6

2 Implementation 8
- 2.1 Registration 8
- 2.2 Communication 8
  - 2.2.1 Put 9
  - 2.2.2 Get 9
  - 2.2.3 Send 10
- 2.3 Synchronization 10
  - 2.3.1 Synchronization order 11
  - 2.3.2 Synchronization points 11

3 Improving the implementation 14
- 3.1 Dynamic request queue allocation 14
- 3.2 Reducing congestion during synchronization 14
- 3.3 Reducing the number of synchronization points 15
  - 3.3.1 Specialized synchronizations 15
  - 3.3.2 Bringing specialized- to general synchronization 16

4 Profiler 18
- 4.1 Data collection 18
- 4.2 Visualization 19
- 4.3 Profiler options 20

5 Real world applications 22
- 5.1 Benchmark 22
- 5.2 LU decomposition 26
- 5.3 Fast Fourier Transform 35

A `C++` code for the parallel LU decomposition 44

Bibliography 53

Preface

The motivation to write this thesis came from the masters course Parallel Algorithms, in which I was introduced to the **B**ulk **S**ynchronous **P**arallel (**BSP**) model. The BSP model is designed around the idea of a BSP computer. A BSP computer is best described as a set of processors that have their own private memory, that is, inaccessible from other processors, and a communication network, which provides a safe way to read or write certain parts of the private memory of other processors. When using the BSP model to write algorithms, this communication network can essentially be viewed as a black box: you do not need to know the internal logic of the network, only how you can use it. For this thesis, we will take a look inside this black box.

The masters course Parallel Algorithms mainly uses MulticoreBSP as an implementation, written in C, which makes it hard, or even impossible to use in combination with the more modern language, `C++`. Another downside is that it adds a lot of repetitive programming tasks, like managing the size of your variables. During the course, we wrote the (extensive) basis of a new implementation in `C++`, Zefiros-BSPLib, that shares many of the same features as MulticoreBSP [1], but had the additional goals of reducing the amount of repetitive programming in the use of the library, as well as cross-platform compatibility. The aim was also compatibility with existing algorithms written using MulticoreBSP . Most of these goals were reached, but due to the difference in compilers for C and `C++`, and different resolutions of timers used, the benchmarks became incomparable between the implementations. This implementation was still a bit rough and a lot of optimization was possible. In this thesis, we will go into detail on the optimizations of the black box on shared memory architectures. Many of the ideas implemented on shared memory architectures can be ported to distributed memory architectures, but again, this is out of scope for this thesis.

On shared memory, all processors could theoretically access the memory of other processors. So why would you need the communication network from BSP to do so? Firstly, you need a safe way to access the memory. You can not just read and write to the memory that is currently used by the other processor, as this could cause undefined behavior in your algorithm. You can not be sure of the state of the variable without the framework to manage it. Secondly, and even more important, if you use the BSP model to design your algorithm, it is easily portable to distributed memory architectures, with even more processors. A shared memory architecture usually comes with a limited number of processors. On distributed architectures, processors and memory are distributed over different (sub) systems. Not all processors have direct access to the same memory. Additional steps need to be taken for communication inside the library, but this will not be discussed in this thesis.

[1]More information on MulticoreBSP can be found in [2].

Chapter 1

The BSP model

This chapter will be an introduction to the **B**ulk **S**ynchronous **P**arallel (**BSP**) model. It gives a brief introduction into the mindset of writing BSP algorithms.

1.1 The model

The BSP model is an idealized model of a parallel computer. The model can be used to structure the parallelization of algorithms in scientific computing, such as the LU decomposition, (sparse) matrix-vector multiplications, etc. The model organizes the algorithms in supersteps, and after each superstep, synchronization happens. This way many needed assumptions can be made about the state of the algorithm, for example when we can be sure the variable contains the desired value from a different processor.

The BSP model distinguishes two important parts of your algorithm: computation and communication. Every algorithm incorporates computation, but when writing parallel algorithms, the intermediate results need to be communicated to other processors before they can be used in further computations. Computation is usually expressed in **flops**, floating point operations. The simplest types of flops are addition, subtraction, multiplication and division. For simplicity, we will assume the cost for each floating point operation is equal. In practice, this is not the case, but we should not complicate our analysis at such a low level. Usually, communication is much more costly than computation, so the focus usually lies on improving communication cost, while still keeping the amount of work balanced between processors.

1.2 Supersteps

Supersteps in a BSP algorithm separate computations on locally known variables from communication. After each superstep, the state of the algorithm on all processors is guaranteed. The supersteps are also used to analyze the cost (in terms of time) of the algorithm. In a computation superstep, each processor computes some result from their private memory. This can be a large computation, like a (partial) matrix vector multiplication of a large matrix and a large vector. After a computation superstep, usually a communication superstep follows to communicate the intermediate result to the processors that need this result in the next computation superstep. Between supersteps, synchronization happens. In a computation superstep, synchronization ensures that each processor is done with the computations. In a communication superstep, synchronization ensures that every processor is done with queuing their

communications, after which the synchronization will make sure the communications will be processed and written to the desired location in the other processors.

In practice, successive computation and communication supersteps can often be combined, because we are usually sure that the computation is finished before the communication is initiated by the processor itself. In our analysis, we will separate them to simplify the analysis of the cost of our algorithms.

1.3 Different types of communication

In the BSP model, there are originally two types of communication possible: writing to another processor, which we will call `Put` from now on, and reading from another processor, which we will call `Get`. Often, the most natural way to think about communication is to `Get`: the processor that needs the information reads it from the other processor. The processor with the information does not have to know who needs the information, he only needs to make it available. When thinking in terms of `Put`, the processor that has the information has to know who will need the information eventually. It might seem less natural, but when we look into the black box of the communication network, we see that `Put` should be preferred over `Get`. In the black box, every communication is queued as a request to the other processor, and `Get` requires an extra step in the internal logic of the synchronization. Moreover, the logic that is written to `Get` information from another processor, can (in many cases) easily be translated to `Put` operations from the other processor.

1.3.1 Put

In order to `Put` information to another processor, the processor that initiates the communication, that wants to write to a different processor, translates the local variable to a global index. Every processor usually has the same variable under the same index. The `Put` request consist of a header, containing this global index, and a copy of the payload itself. During the synchronization, the target processor translates the global index back to its own local variable, and writes the payload to this location. Synchronization ensures that the variable in the other processor contains the local information at the beginning of the next superstep.

1.3.2 Get

In order to `Get` information from another processor, we need to do a little more work. The processor that initiates the communication, that wants to read from a different processor, translates the local variable to a global index. The `Get` request consists of just the header, with the global index. During the synchronization, the target processor processes this header, queuing another request. This new request is quite similar to the put request, containing the global index from the `Get` request and a copy of the local variable as payload. Then, in the same synchronization, the processor that queued the `Get` request in the first place writes the payload to its local destination variable. Synchronization ensures that the variable contains the information from the desired location in the other processor at the beginning of the next superstep.

Since we need an extra request for `Get`, the speed of `Get` will be slightly worse than the speed of `Put`. In the cases that the algorithm can be written just

as easily using `Put`, it is recommended to do it that way. In many cases, both the target and the source of the request know which information needs to be communicated. However, `Get` is still useful in the cases that only the processor that initiates the `Get` communication knows which information is required. For example, the communication could be conditional, depending om some state that is only known to the processor that wants to `Get` information. In order to rewrite the algorithm to use `Put`, we would need to communicate the conditional value depending on the state of the target processor before we can initiate `Put`. Aside from complicating your algorithm, this would require an extra superstep, which usually has a higher cost penalty than using `Get` over `Put`. In these cases, `Get` is preferred over `Put`.

1.3.3 Send

In an extension of the BSP model, a third type of communication is added: sending an information payload with a tag, also called a label, to a different processor. This way, the variable does not have to exist under a global index, but the receiving processor needs to know how to translate the tag to the desired destination for the payload. This is very similar to Message Passing, which is a different model for writing parallel algorithms, but the difference is that in BSP, the sending of messages happens in bulk: everyone passes messages at the same time and synchronizes in bulk. In message passing, it happens pairwise and synchronization is also in pairs, not in bulk. `Send` introduces much of the same logic that can be applied in message passing, but still has the mindset of bulk synchronization.

Peeking into the black box of Send, synchronization is a bit more difficult. For Send, synchronization should ensure that the message that is sent is in a queue, accessible from the target processor, at the beginning of the next superstep, and the queue for new requests is usable in the same superstep. What this means internally for the communication network, is that when the `Send` is performed, the tag and a payload containing a copy of the information are queued at the sending processor. During the synchronization, the tag and the payload are copied to a queue that will be accessible in the next superstep for the receiving processor, and the `Send` queue at the sending processor is cleared. This means that the payload will be copied twice: once at the source processor to the queue, and once from the queue to another queue, accessible from the target processor. For larger payloads, this becomes very inefficient.

Additionally, the size of the tag should be synchronized between the processors. `Send` should only be used if a global index for the variable becomes nearly impossible, or if this severely simplifies the logic in the algorithm. For `Send` to become useful, the implementation of the `Send` primitive in the library should be restructured.

1.3.4 Registration of variables

As mentioned, variables used as target for communication need to be registered globally before they can be used. Registration of variables is called `Push`, you push it onto a stack. De-registration is called `Pop`, you pop it from the stack. In both MulticoreBSP and Zefiros-BSPLib, it is not a pure stack, because you can `Pop` variables in a different order than the order of `Push`.

1.4 Synchronization

The communication part of the BSP model is quite straightforward. Nothing too complicated has happened yet, aside from how the synchronization happens internally. Synchronization is therefore the biggest part of the black box of the communication network. Details will be explained later, but the difficult part of the synchronization is that all three types of communication, together with registering and de-registering of variables, can happen in the same superstep. All of these have to happen in a certain order to ensure the correct behavior of the BSP algorithms.

1.5 The BSP cost

The time analysis of any BSP algorithm can be expressed as the BSP cost. The BSP cost takes the cost of communication and synchronization into account. The cost of communication and synchronization will be different for different machines, and even more for different architectures. The cost of communication and synchronization on shared memory architectures will be considerably less than it is on distributed systems, where communication and synchronization goes over the (local) network, for example via TCP. The parameters g and l are to be determined for the BSP computer you are going to use. A way to get an approximation will be discussed later, in section 5.1.

To analyze the cost of an algorithm as generically as possible, we separate the BSP cost into three parts. The *computation cost*, which is the approximate number of floating point operations (**flops**) of the computation on each processor. The number of flops should be nearly equivalent for every processor, but in the case it is not, the maximum number is taken for the BSP cost, as the other processor will have to wait for this processor to finish its computations. Computation is expressed as an integer number of flops. The *communication cost* is computed by multiplying the amount of real numbers communicated by the **communication cost parameter** g. This g is different for different BSP computers. Again the maximum number of communications is taken for the computation of the BSP cost. Note that both incoming and outgoing communication should be taken into account. Finally, we have the *synchronization cost*. This is computed by multiplying the number of supersteps in the algorithm by the **synchronization cost parameter** l. The parameters g and l are approximations of the amount of time it takes to communicate or synchronize, multiplied by the number of flops per second the BSP computer can perform.

It is often useful to analyze the cost per superstep. The *superstep cost* is then

$$w + hg + l,$$

where $w = \max_{0 \leq s < p} w^{(s)}$ and $h = \max_{0 \leq s < p} \left(\max\{h^{(s)}_{\text{send}}, h^{(s)}_{\text{receive}}\} \right)$, where $w^{(s)}$ and $h^{(s)}$ are the number of flops and the number of communicated real number respectively for processor s.

The total cost of the BSP algorithm then becomes

$$W + Hg + Sl = \sum_{i=1}^{S}(w_i + h_i g + l) = \sum_{i=1}^{S} w_i + g\sum_{i=1}^{S} h_i + Sl,$$

where W, H, S are usually written as functions of the problem size, and written in terms of $\left\lceil \frac{n}{p} \right\rceil$. Another important terminology in the analysis of the BSP cost is the ***h*-relation**. An h-relation is a communication superstep, where h is as before, so each processor sends and receives at most h real numbers. A **full** h-relation is an h-relation where each processor sends and receives exactly h real numbers. The cost of an h-relation reduces to $hg + l$, because we do not have computation here.

1.6 Example BSP algorithm

An algorithm that is commonly used to introduce BSP algorithms, is the inner product of two vectors. For the sequential case, the algorithm is described by Algorithm 1.1. It is just the sum of the element-wise products. To transform the algorithm to a BSP algorithm, we want to split this sum over the processors. In this example, it is quite easy, because the order in which the elements are summed up is not important. Different types of distributions of data are possible. The most general distributions are block distributions, cyclic distributions and block-cyclic distributions. The block distribution splits the data into p contiguous blocks, p being the number of processors used in the algorithm. Cyclic distribution assigns every element with index $s+k\cdot p$ to processor s, where $0 \leq s < p$, where p is again the number of processors used in the algorithm. The block-cyclic distribution is a hybrid between the two previous: it separates the dataset into blocks of size $q < \left\lceil \frac{n}{p} \right\rceil$, and the blocks are assigned to processors like elements were in the cyclic distribution.

For the inner product algorithm, any distribution would be suitable. For the sake of simplicity, the cyclic distribution is used. The cyclic distribution can also be described by a function ϕ mapping the index of an element i to the processor with number t, $P(t)$. For the cyclic distribution, this function is $\phi(i) = i \mod p$. Every processor computes the partial inner product for the indices assigned to the processor, in a computation superstep. Then, in a communication superstep, every processor communicates its partial inner product to all other processors. Finally, every processor computes the sum of all partial inner products to compute the full inner product. The pseudo code is in Algorithm 1.2. The notation $P(t)$ is an enumeration of the processors, with $0 \leq t < N$, where N is the number of used processors.

Of course, instead of communicating the partial inner product to every processor, you could communicate the partial inner product from all processors to processor 0, and let processor 0 compute the full inner product. This might seem like it would be less work, but because of the parallelism, the cost would be the same if we were to send the information to every other processor, instead of just to processor 0. The maximum of the received and sent communication does not change. While processor 0 is computing the sum of the partial inner products, the other products are idle, so why not compute it on all processors simultaneously. The latter is the mindset of BSP algorithms: every processor eventually has the result of the computation, so that it can be used as part of a bigger algorithm. If we would just compute the final sum on

processor 0, we would need an extra communication superstep to distribute the final sum again, resulting in a higher BSP cost.

Algorithm 1.1: Sequential inner product algorithm[1].

Input : $\mathbf{x}, \mathbf{y}$: vector of length n.
Output: $\alpha = \mathbf{x}^T \mathbf{y}$.
$\alpha := 0$;
for $i := 0$ **to** $n - 1$ **do**
  $\alpha := \alpha + x_i y_i$;

Algorithm 1.2: Inner product algorithm for processor $P(s)$, $0 \leq s < p$[1].

Input : $\mathbf{x}, \mathbf{y}$: vector of length n,
distr($\mathbf{x}$) = distr($\mathbf{y}$) = ϕ,
with $\phi(i) = i \mod p$, for $0 \leq i < n$.
Output: $\alpha = \mathbf{x}^T \mathbf{y}$.
(0) $\alpha_s := 0$;
for $i := s$ **to** $n - 1$ **step** p **do**
  $\alpha_s := \alpha_s + x_i y_i$;
(1) **for** $t := 0$ **to** $p - 1$ **do**
  put α_s in $P(t)$;
(2) $\alpha := 0$;
for $t := 0$ **to** $p - 1$ **do**
  $\alpha := \alpha + \alpha_t$;

[1] These algorithms are reconstructed from [1].

Chapter 2

Implementation

This chapter describes the work done prior to the start of this thesis. For the shared memory implementation Zefiros-BSPLib, we need a way to separate the processors data, but we need to share some data in order to communicate. Here, we make a clear distinction between shared memory and private memory. Private memory contains all variables allocated at a certain processor. When we register a variable with `Push`, the variable is accessible via the shared memory of all processors. The data for communication is all written in shared memory by the library.

2.1 Registration

In order to communicate, we first need to register the variable that we want to share with other processors. We do this by registering the pointer to the variable and the size in bytes. To make this mapping, we will keep track of a stack S and a map M. The pointer p is added to S at the next available index, i. Then, in the map we store an object containing this index and the size, so it can be viewed as a map $M : p \mapsto (i_p, \text{size}_p)$. The stack can also be viewed as a map $S : i_p \mapsto p$. New entries can be added to the map by `Push`, and entries can be removed by `Pop`. These are regular function names in the context of a stack. The stack that we use here is not a pure stack, in the sense that for a pure stack, we could only remove the top element. Here, we can `Pop` the variables in the stack in any order. This data structure was easily implementable with the `C++` data structures `std::vector` for the stack, and `std::map` for the map.

2.2 Communication

Communication consists of two parts: header and payload. This is true for any type of communication. The headers are stored in a queue, and the payload is stored in a `StackAllocator`. This is a data structure we came up with to efficiently manage the storage of multiple payloads. The `StackAllocator` is a stack of bytes, in which we can allocate blocks of arbitrary sizes. This way, we can store payloads of any size in contiguous memory, without preallocation of a fixed size buffer. After communication, the stack cursor is reset to the bottom of the stack. The allocated size does not change. When allocating a block after that, the existing information above the cursor is overwritten. The `StackAllocator` tests whether the block fits in the stack. If not, the stack is resized so that the new block will fit. The resize is done with a factor 1.6, close

to the golden ratio[1]. Upon allocation of such a block, the location of the cursor at the beginning of the data is returned. The internal logic of the different types of communication will now be discussed in more detail. When the library is compiled in debug mode, the stack is filled with random data upon reset, to also test if the information is overwritten correctly. Otherwise, old data might be reused without us noticing. If the random data is used, we know there is something wrong with the improvements made to the library.

2.2.1 Put

Before calling `Put`, the variable p has to be registered in a previous superstep, by the processor that makes the request. We assume p' is registered under the same index $i_{p'} = i_p$ by the target processor. For `Put`, the header is of the form

$$(i_p, \texttt{size}, \texttt{offset}, \texttt{stack location}).$$

Upon calling `Put`, the information is copied from the source and added to the stack allocator. The pointer p is mapped to i_p, and the header is added to the queue. The request queues are separated for each targeted processor. Because we work on shared memory, the stack allocator can be shared for all target processors. The stack cursor ensures each processor can retrieve the right data. The shared stack allocator prevents memory fragmentation, and reduces the number of resizes needed to fit the different payloads. During the synchronization, every processor looks at the shared memory of the other processors, and processes the `Put` request queue targeted at the processor. While it processes the queue, it extracts the information from the stack allocator of the source processor. The global index $i_p = i_{p'}$ is mapped to the pointer p' local to the target processor, and the payload is then written at the offset that was in the header. After everyone is ready, the stack allocator cursor is reset, without shrinking the stack.

2.2.2 Get

Before calling `Get`, the variable p has to be registered in a previous superstep, by the processor that makes the request. We assume p' is registered under the same index $i_{p'} = i_p$. The variable q does not necessarily have to be registered, but it is allowed. For `Get`, the header is of the form

$$(i_p, \texttt{size}, \texttt{offset}, q).$$

Upon calling `Get`, only the request header is queued. Then during the synchronization, all processors clear their own *get* stack allocator, and look at the shared memory of the other processors. If a `Get` request is queued to the processor, it makes another request itself, a *buffered get request*, with a header of the form

$$(q, \texttt{size}, \texttt{stack location}),$$

targeted at the processor that made the `Get` request. They map the global index $i_p = i_{p'}$ to their local pointer p', then copy the payload from p', starting

[1] More on the golden ratio is at http://mathworld.wolfram.com/GoldenRatio.html. In theory, this works better than rescaling with a factor 2, because memory will be better reusable for re-allocation as explained in https://crntaylor.wordpress.com/2011/07/15/optimal-memory-reallocation-and-the-golden-ratio/.

at the offset that was in the header, and allocate it in the stack allocator for the buffered `Get` request with the requested size. The stack cursor that is returned from the stack allocator, together with the pointer q and the size from the header, both unchanged, are queued as a buffered `Get` request. This is all done in the shared memory of the target of the `Get` request. After all `Get` requests are buffered, every processor again looks at the memory of the other processors for the buffered `Get` requests. Every processor then copies the payload to their own destination q. The variable q does not necessarily have to be known by the other processor, but can be seen as an identifier for the `Get` request. Without this q, we would need a second mapping from $q \mapsto j_q$ and $j_q \mapsto q$. This is not necessary with this construction.

2.2.3 Send

For `Send`, the header is a little different. No mapping of pointer p to i_p is needed. Instead, `Send` requests are accompanied by a tag. The tag itself is similar to the payload, but often has a much smaller size. The header is of the form

$$(\texttt{payload stack cursor}, \texttt{size}_{\texttt{payload}}, \texttt{tag stack cursor}, \texttt{size}_{\texttt{tag}}).$$

Upon calling `Send`, the tag and the payload are copied to a stack allocator, and the returned indices, together with the sizes, are queued in the shared data of the sending processor. The tag data and payload are written in the same stack allocator. During synchronization, every processor looks at the shared data of the other processors, and merges the `Send` request queues and stack allocators to its own shared data, into a single queue and a single stack allocator. The queues and stack allocators of the sending processors are then cleared. This merged queue is then available in the next superstep with `GetTag` and `Move`. `GetTag` retrieves the tag data. After the user has decided what to do with the data belonging to this tag, the data can be retrieved with `Move`. At the end of the superstep, the merged queue and stack allocator are cleared.

2.3 Synchronization

Every type of communication and every variable (de-)registration is only queued before we start to synchronize. Synchronization ensures everything is delivered in the expected order and with the expected value. Synchronization ensures that everyone is done with this superstep, and that after the synchronization all `Get` requests are retrieved from the target processor, `Put` requests are written in the target processor, variables are properly registered or deregistered, and `Send` requests are available in the target processor at the beginning of the next superstep. Internally, the synchronization also ensures that the queues and stack allocators are properly cleared. Synchronization is often viewed as a barrier. After this barrier, the state of the algorithm is again ensured. Internally however, multiple barriers are needed to ensure the proper order of synchronization.

2.3.1 Synchronization order

Different types of communication can happen in the same superstep, and the order in which they are processed is very important. To ensure they are handled in the proper order over all processors, the synchronization contains internal **synchronization points**. Synchronization points ensure that all processors are at the same stage of synchronization. Synchronization points can be viewed as internal barriers for the synchronization. The restrictions on the order of synchronization are as follows:

i. Every processor needs to be completely finished with computations before `Get` requests are buffered. Otherwise, older intermediate values could be buffered.

ii. Every processor needs to be finished with computation before `Put` requests are processed. Otherwise, values could be overwritten before the target processor is done with its computation on this variable.

iii. Buffering of `Get` requests has to be completed on all processors before processing the buffered `Get` requests and before processing the Put requests. The former is immediately clear. The latter is necessary, because otherwise values written from the `Put` requests could be buffered as `Get` requests.

iv. Tag size needs to be synchronized over all processors before processing `Send` requests.

v. `Pop`, `Put` and (buffered) `Get` request need to be completely processed on all processors before `Push` requests are processed. This ensures that the new variables from `Push` requests can not be accessed too early on.

vi. `Send` requests need to be completely processed into the receiver queue on all processors before clearing the sender queues.

vii. `Put` requests need to be processed on al processors before clearing `Put` payload buffers.

viii. `Put` buffers can be cleared at the beginning of the next superstep without internal synchronization points.

ix. `Push` requests need to be finished before the beginning of the next superstep.

Taking these restrictions into account, the pseudo-code in Algorithm 2.1 describes the order of the synchronization and the minimal number of internal synchronization points. We will call the internal synchronization points `SyncPoint` in the pseudo-code. The `ClearPutBuffers()` could be placed after the last `SyncPoint`, but this way the synchronization will end more simultaneously, and it does not really matter for the performance.

2.3.2 Synchronization points

As mentioned, these internal synchronization points can be viewed as barriers. With `C++`, two simple barriers are easily implementable.

Algorithm 2.1: Synchronization pseudo-code

```
1  function Sync()
2      // This ensures i.  and ii.
3      SyncPoint();
4      ProcessTagSizeUpdate();
5      BufferGetRequests();
6      // This ensures iii.  and iv.
7      SyncPoint();
8      ProcessPopRequests();
9      ProcessSendRequests();
10     ProcessPutRequests();
11     ProcessGetRequests();
12     // This ensures v., vi.  and vii.
13     SyncPoint();
14     ClearSendRequests();
15     ClearPutBuffers();
16     ProcessPushRequests();
17     // This ensures ix.
18     SyncPoint();
19 end function
```

The first is a *spin barrier*. Each processor reads the current generation number, and then decreases a counter. After they decreased the counter, they continuously reread the generation number, until the generation number is increased. The last processor to reach the counter decreases it to 0, and resets the counter. After the counter is reset, the processor increases the generation so that the other processors can also continue. `std::atomic_uint_fast32_t` is used for the counter and the generation number, which means operations such as read, write, increment, decrement, add, subtract happen atomically, that is, we can ensure that each of these operations is completed before the next happens. The most important operation for the spin lock is the decrement-and-read of the atomic variable. This operation decrements the counter and returns the value afterwards, all before the next operation can happen. This way, we can ensure that every processor gets the correct counter value. It is called a spin barrier, because every processor *spins* on the check whether the generation has changed.

The second is a *condition variable barrier*. A condition variable has a function `wait`, in which a condition is checked. The condition in this case being the change of generation number. The condition variable barrier has a similar counter as the spin barrier. If the decremented value is nonzero, `wait` is called. This triggers the operating system to put the process on hold, and to release the processor. When the counter hits zero and the last processor has entered the barrier, it calls a `notify_all` to wake up all threads again. The upside of the condition variable barrier is that it relieves the processor of the stress of computations, whereas the spin barrier constantly keeps comparing the generation numbers. Another upside is that we can now test the correctness of our algorithm for more processors than the machine actually possesses: because the process is put on hold, another process can occupy the same processor, thus we can emulate a system with more processors. The spin barrier claims the processor for itself. The downside of a condition variable barrier

is that the overhead of putting the thread on hold is quite large, as this has effects all the way down to the scheduler of the operating system. This makes the synchronization much more costly, as we have multiple synchronization points in one synchronization.

For this purpose, a third option is suggested, a *mixed barrier*. The mixed barrier, as the name suggests, is a mix of the spin barrier and the condition variable barrier. The counter and generation work just like the spin barrier. The processor first enters the spin barrier. Instead of possibly spinning indefinitely, the processor spins until it has reached a predefined number of iterations. The spinning would stop earlier if the generation number changes. If however, the maximum number of iterations is reached and the generation has not changed, the processor will `wait` on the condition variable. Once the counter hits zero, the generation is increased first, and then all processors that are waiting for the condition variable will be notified by `notify_all`. This way, if some are spinning and some are waiting for the condition variable, everyone will continue. The upside of this barrier is that in case the algorithm is truly balanced in both computation and communication, no processor will ever reach the condition variable, so we will not have the overhead of the putting on hold by the operating system scheduler. Another upside is that we can still emulate a system with more processors than the physically available processors. A downside is that, even though the algorithm is truly balanced, background interference from the system may still sometimes claim a processor for a number of iterations. This could cause one of the processors to enter the spin barrier slightly delayed, and all other processors to enter the condition variable barrier. Due to the wake up time, it could cost some synchronization to re-balance barrier entry times, slowing down the algorithm. That being said, the benchmark in section 5.1 does not suffer from this background interference on the machine it was tested on. Algorithms with larger supersteps are more likely to suffer from overhead of the condition variable barrier, as small interruptions that did not affect small supersteps may now stack up to cause a large enough delay.

Chapter 3

Improving the implementation

The first version of the implementation has now been discussed. As it was a first version, a lot of optimization was possible. These improvements were done as part of this thesis. Some of these improvements are aimed both at improving communication and synchronization cost.

3.1 Dynamic request queue allocation

In the first version of the library, request queues were completely cleared, freeing up the allocated memory, and allocating new memory as needed. This turned out to add a lot of time overhead to the communication, as well as synchronization time. In order to keep the flexibility of the dynamic allocation of memory, but still reduce the overhead from queue resizes, a queue similar to the stack allocator is introduced. Instead of clearing the queue and freeing the memory, the head and the tail of the queue are reset to the first element in the queue. New requests overwrite older requests, and resizing of the allocated memory for the queue again happens with a factor 1.6. This way, the overhead for resize is minimized to the few occasions that the new queue of size 1.6 times the old size is exceeded. In general program such as the FFT or the LU decomposition, communication volume stays the same or even decreases over the course of the algorithm. This causes the overhead of resizing only to appear in the early stages of the algorithm. This new request queue has a generalized implementation for all `Put`, `Get`, `Send`, `Push` and `Pop` requests. Not only does it reduce the communication time, but also the synchronization time, as the memory is not freed during synchronization anymore.

3.2 Reducing congestion during synchronization

In the first version of the library, every processor had the same order in which the communication was handled from the other processors, namely $0, 1, ..., p-1$. This caused congestion at the shared memory of the processors, because every processor was accessing it at the same time. Instead of starting at 0, we could also start at our own processor number s, so the order would become $s, s+1, ..., p-1, 0, 1, ..., s-1$. Different ways of ordering could be thought of, and this idea was generalized in [3], where it is referred to as a *Latin Square*. A Latin square is an $n \times n$ matrix, filled with n different symbols. Each symbol occurs exactly once in every row, and once in every column. The larger n is, the more variations of the Latin square exist. Any Latin square could be used, but in the most general case, the example above should satisfy. This is also the most simple one to implement in code. The example presented above is easily implemented by duplicating the for loop and changing the bounds. As

this would almost double the amount of code, and because it would make the code unmaintainable, this is not very desirable. This is where the C++ templates and lambda functions come in handy. The library now contains a function

$$\texttt{BSPUtil::SplitFor}(start, split, end, body),$$

where *body* is a lambda function with the loop iterator as argument. The loop can then be written as follows.

```
1 BSPUtil::SplitFor( 0, s, p, [&] ( uint32_t i )
2 {
3     // Process requests from processor i to me
4 } );
```

This way, we can write the for loop quite naturally, and the C++ lambdas do not affect performance. We can use this for-loop to improve every type of synchronization of communication. The split for-loop is only used internally, but could also be used for other applications.

3.3 Reducing the number of synchronization points

The synchronization currently needs four internal synchronization points, because some types of communication or registration need to be performed before all others. But in a general superstep, not every type of communication is actually used. For this purpose, we introduce the specialized synchronization interface.

3.3.1 Specialized synchronizations

The specialized synchronization interface consists of

$$\texttt{SyncPutRequests}, \texttt{SyncGetRequests}, \texttt{SyncSendRequests}.$$

Turns out, we can do most of these specialized synchronizations with only two internal synchronization points. That is possibly a 50% reduction of the synchronization parameter l.

For `Put`, we need one internal synchronization point to ensure everyone is done. Then we can process the `Put` requests. After the `Put` requests are completed, we need another one to ensure all `Put` requests have been processed so that is another synchronization point. Finally, the buffers can be cleared without synchronization, as this happens locally.

For `Get`, we also need to ensure everyone is done, so that is one internal synchronization point. After that, we clear the buffers and add the `Get` requests to the buffer. We need another internal synchronization point to ensure everyone is done buffering. After that, the buffered `Get` requests can be processed without further synchronization, because the buffers would only be cleared after the first synchronization point in the next synchronization.

For `Send`, we also need an internal synchronization point to ensure everyone is done. Then we can merge the queues and stack allocators to the single queue and single stack allocator needed in the next superstep. This requires another synchronization point to ensure everyone is done. We can then safely clear the send buffers and begin the next superstep without further synchronization points.

This way, we have reduced the number of internal synchronization points to two instead of four for every type of communication. We could do something similar for registration and de-registration, but these often only happen during initialization and at the end of the algorithm, so this is probably not worth the trouble. Combined specialized synchronizations could be introduced, but are deprecated by the following section.

3.3.2 Bringing specialized- to general synchronization

For the specialized synchronizations, the user needs to check that he only has a certain kind of communication during the superstep to be able to make use of specialized synchronizations. This can be bothersome, as algorithms can improve over time, and different types of communication can be introduced to improve the algorithm. Having to check the correctness of the chosen specialized synchronizations is prone to errors. It would be better to incorporate this in the general synchronization. The solution is quite straightforward.

Before the first synchronization point, every processor looks at its own shared memory and keeps track of booleans for which type of communication or registration has happened. This happens only once at the beginning of the synchronization, not for every call to communication functions. These booleans are stored in shared memory. Immediately after the first synchronization point, the booleans are merged with an `or` operation over all processors. This ensures that every processor knows of the global presence of all types of communication and registration. If globally there exists a request of a certain type, every processor will enter the synchronization points corresponding to that type of request. This way, we have the power of the specialized synchronization points with minimal overhead for the computation and merging of the synchronization booleans. The improved synchronization is shown in Algorithm 3.1.

This does not completely deprecate the specialized synchronizations. If the user is absolutely sure the specialized synchronization can be used, this prevents the overhead of computing and merging the synchronization booleans. This is especially useful when a large number of processors is used.

Algorithm 3.1: Improved synchronization pseudo-code

```
1  function Sync()
2      ComputeSyncBooleans();
3      // This ensures i. and ii.
4      SyncPoint();
5      MergeSyncBooleans();
6      ProcessTagSizeUpdate();
7      BufferGetRequests();
8      if syncBools.hasTagSizeUpdate or syncBools.hasGetRequests then
9          // This ensures iii. and iv.
10         SyncPoint();
11     ProcessPopRequests();
12     ProcessSendRequests();
13     ProcessPutRequests();
14     ProcessGetRequests();
15     if syncBools.hasSendRequests or syncBools.hasPopRequests or
16         syncBools.hasPutRequests or syncBools.hasGetRequests then
17         // This ensures v., vi. and vii.
18         SyncPoint();
19     ClearSendRequests();
20     ClearPutBuffers();
21     ProcessPushRequests();
22     if syncBools.hasPushRequests or syncBools.nothing then
23         // This ensures ix.
24         SyncPoint();
25 end function
```

Chapter 4

Profiler

To improve on an algorithm, it is often useful to have some visualization of the communication pattern, and the actual timing of each superstep. As mentioned, we ignored some terms in the analysis of the cost of a BSP algorithm. Under normal circumstances, the analysis is quite accurate, but sometimes, the result might still be very different to the expectations. Without proper visualizations, it can be very cumbersome for the user to write their own profiler for every separate algorithm, over and over. For this purpose, a profiler is included with Zefiros-BSPLib.

4.1 Data collection

In order to give useful output, we need to collect several types of data. We need communication time, computation time and synchronization time for separate supersteps. We need the size and number of the payloads for separate supersteps. Superstep numbers can also be very useful, to give an average of the time each superstep takes in terms of computation and communication and synchronization. The distinction between communication and synchronization is a bit of gray area. Large parts of communication routines happen during synchronization. Therefore, these two are usually most useful added together, instead of separately.

At the beginning of each superstep, a timer is started. This keeps track of the time of the entire superstep. At the beginning of each communication, another timer is started. At the end of the communication, the timer is stopped and the elapsed time is added to the communication time of the superstep. At the beginning of the synchronization, the communication timer is started again, and at the end of the synchronization, the synchronization time is also added to the communication time. The superstep timer is stopped, and the communication and synchronization times are subtracted from the superstep timer to get the computation time.

A manual override is also possible, to manually start and stop the communication timer. This is useful when there is a large communication loop. The loop introduces some overhead, which is not really computation, but would otherwise be counted as computation. We can simply put

```
1 BSPProf::InitCommunication();
2 // Loop
3 BSPProf::FinishCommunication();
```

around the loop, and the library handles the rest. Timings for command line input from the user are not very useful to include in the profiler timings, as this would cost much more than the average superstep. In order to avoid recording these timings, we can `PauseRecording()` and `ResumeRecording()` as

needed to exclude it from the profiler. Supersteps can be marked in two ways: `MarkSuperstep()` simply increases the superstep counter.
`MarkSuperstep(i)` resets the superstep counter to i. This way, if we have a loop containing multiple supersteps, we can categorize them by superstep number.

Finally, we need to count the number of incoming and outgoing requests for each processor, and compute the accumulated size of incoming and outgoing communication for each processor. This is done immediately after the first synchronization point. If there is actual data to be collected, we need another synchronization point right after the data collection, as other processors might start to manipulate the data otherwise. This is necessary in the current version, because every processor also looks at the data of other processors to collect its own profiler data. In a next version, it would be better to only look at requests originating from the processor itself, and store it in a sender-receiver matrix of size $p \times p$, to later add up to a total received volume.

Currently there is already a form of matrix data of size $p \times p$, but this matrix is split into rows. Each processor contains a row with information about how much he is sending to each of the other processors. To get a complete picture of this information, the processor needs to look at the `Get` requests of all other processors. Instead of keeping track of a single row, each processor could also keep track of both a column and a row: a column with the amount of received information of `Get` requests that the processor itself has queued, and a row with sent information from `Send` and `Put` requests. These queues are all guaranteed to be finished before the processor itself enters the synchronization. The rows of each of the processors can then be concatenated to form a $p \times p$ matrix, and then each of the columns can be added to the columns of the matrix to complete the information. Textually, this is easy to grasp, but when it comes to thread-safe data collection, this is slightly harder than it seems, but could be implemented in a next version. The advantage of only collecting data from the queues of the processor itself, is that it can be done before the first synchronization point, and no extra synchronization will be needed after data collection.

4.2 Visualization

The profiler has quite some built in visualization options.

i) A stacked bar plot with on the horizontal axis the elapsed time and on the vertical axis the maximum number of bytes sent or received by a processor. This corresponds to the $h^{(s)}$ in our cost analysis. On the horizontal axis, the width of the bars is communication+synchronization time. The width of the gaps is computation time. The height of each of the parts of the stacked bars represents the size for a specific processor. This type of visualization is similar to the visualizations in the Oxfort BSP toolset profiler [4]. Each of the individual parts is connected by a dashed line, to make it a little more readable, this is the addition made by Zefiros-BSPLib. Also, Zefiros-BSPLib does not separately plot sending and receiving information. Instead, the maximum of the two is plotted. An example of this in the following chapter, in Figure 5.4.

ii) This is similar to the previous plot, but instead of the size in bytes, the number of requests is reported. An example of this in the following chapter, in Figure 5.4.

iii) This is similar to the first item. Again, this is a stacked bar plot with the size in bytes per processor. Instead of the time on the horizontal axis, the superstep numbers are shown under the bar, and the width does not have any specific meaning. The bars are grouped by superstep number, and a confidence interval is added to the tops of the bars to show the minimum and maximum size for that superstep for each of the processors. An example of this in the following chapter, in Figure 5.5.

iv) This is similar to the previous plot, but instead of the size in bytes, the number of requests is again reported. An example of this in the following chapter, in Figure 5.5.

v) This is another bar plot, this time also categorized by superstep number on the horizontal axis. This bar plot has two colors stacked on top of each other, the top for communication and synchronization, the bottom for computation. Each superstep has p such double bars. From this plot, imbalance in either communication or computation time can be read, and the ratio between computation and communication can be seen for each superstep. Confidence intervals are added to the tops of the bars, to indicate the maximum and minimum time needed for that specific superstep by the processor. An example of this in the following chapter, in Figure 5.6.

vi) This plot contains the ratio

$$\texttt{time}_{\texttt{communication}}/(\texttt{time}_{\texttt{computation + synchronization}}),$$

plotted for each superstep, not categorized but in chronological order. This is a more readable plot of the ratio, and can be used to analyze if the BSP algorithm is communication or computation bound, and if the behavior changes over the course of the algorithm. An example of this in the following chapter, in Figure 5.8.

vii) The final visualization is a plot of the matrices described before. For each superstep, such a matrix is plotted. The rows indicate the sending processor, the column contains the receiving processor. Each matrix entry contains the number of bytes sent from the row processor to the column processor. This is visualized by a heat map: increasingly dark color means more communication. An example of this in the following chapter, in Figure 5.7. Communication patterns are visualized in this plot.

These were all the useful plots that came to mind. Since the data collection and visualization is accessible to the user, more plots could be added later when needed.

4.3 Profiler options

A profiler is nice during development of the algorithm, but in production, we do not want to show the profiler each run of a certain algorithm. For this

purpose, the profiler is opt-in. The profiler can easily be substituted by a profiler with less options, for example a profiler without the matrix plots. An entirely different profiler could also easily be implemented and used by the library, without changing the code of the library. As the profiler is opt-in, the default profiler is a `VoidRecorder`. This is a special profiler with empty functions. The `C++` compiler has dead code removal, so empty functions will be optimized such that there is no call to the empty function in the compiled program. The different parts of the profiler can easily be turned on or off, as will be explained in future documentation of the library.

Chapter 5

Real world applications

Now that we have discussed the implementation and improvements of the Zefiros-BSPLib library, we can test it in some real world applications, to see how larger algorithms can be designed as a BSP algorithm. As discussed in chapter 1, the communication parameter g and the synchronization parameter l are to be determined per architecture. For this purpose, the `bspbench` benchmarking tool was written for the `BSPedupack`, a library for educational purposes, also containing some examples of portable BSP algorithms, like the Fast Fourrier Transform and the LU decomposition, which will be discussed in the next sections.

5.1 Benchmark

The `bspbench` utility was written to benchmark a generic BSP library that adheres to the BSP interface, on a generic BSP computer. This program outputs the communication and synchronization parameters in terms of flops. This is done by first measuring the approximate number of floating point operations that can be performed per second by the BSP computer. After that, h-relations are performed for $1 \leq h \leq h_{max}$, and the time per h-relation is measured. This time is, like the cost of an h-relation, assumed to be linear with respect to h. This seems to be accurate, at least for shared memory architectures. The data collected from the h-relations is then processed by a **Least Squares** algorithm to approximate the parameters g and l. We will be using the **O**rdinary **L**east **S**quares (**OLS**) algorithm. The OLS algorithm gives the best estimation for b in $y_i = bx_i + \epsilon_i$, that is, the sum of squared residuals $S(b) = \sum_{i=1}^{n}(y_i - bx_i)^2 = \sum_{i=1}^{n} \epsilon_i^2$ is minimized by the algorithm. We could also write this in vector notation, which would give $S(b) = \|\mathbf{y} - b\mathbf{x}\|_2^2$, where $\|\cdot\|_2 = \sqrt{\langle \cdot, \cdot \rangle}$ and $\langle x, y \rangle$ is the dot-product of x and y. Then

$$\hat{\beta} = \arg\min_{b \in \mathbb{R}} S(b) = \frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\|\mathbf{x}\|_2^2}.$$

This is proven in the following lemma.

Lemma 5.1.1. Let $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$.The OLS linear estimator

$$\hat{\beta} = \frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\|\mathbf{x}\|_2^2}$$

minimizes $S(b) = \sum_{i=1}^{n}(y_i - bx_i)^2$.

Proof. We begin this proof with rewriting $S(b)$ and collecting terms of b.

$$\begin{aligned} S(b) &= \sum_{i=1}^{n}(y_i - bx_i)^2 = \|\mathbf{y} - b\mathbf{x}\|_2^2 = \langle \mathbf{y} - b\mathbf{x}, \mathbf{y} - b\mathbf{x}\rangle \\ &= \langle \mathbf{y}, \mathbf{y}\rangle - b\,\langle \mathbf{x}, \mathbf{y}\rangle - b\,\langle \mathbf{y}, \mathbf{x}\rangle + b^2\,\langle \mathbf{x}, \mathbf{x}\rangle \\ &= \|\mathbf{y}\|_2^2 - 2b\,\langle \mathbf{x}, \mathbf{y}\rangle + b^2\,\|\mathbf{x}\|_2^2\,. \end{aligned}$$

Since this is a quadratic equation and the coefficient in front of b^2 is positive, we know that the global minimum of this function is located at the value of b for which $\frac{\partial}{\partial b}S(b) = 0$. The $\hat{\beta}$ satisfying this condition should satisfy

$$\begin{aligned} 0 &= \frac{\partial}{\partial b}S(b)\bigg|_{b=\hat{\beta}} = \frac{\partial}{\partial b}\left(\|\mathbf{y}\|_2^2 - 2b\,\langle \mathbf{x}, \mathbf{y}\rangle + b^2\,\|\mathbf{x}\|_2^2\right)\bigg|_{b=\hat{\beta}} \\ &= -2\,\langle x, y\rangle + 2\hat{\beta}\,\|x\|_2^2\,, \end{aligned}$$

which is equivalent to

$$\hat{\beta} = \frac{\langle x, y\rangle}{\|x\|_2^2}.$$

□

Suppose we would now run the benchmark and collect t_h for h-relations with $h \in h_0, h_0 + 1, ..., h_1 - 1, h_1$. We now have a way to compute the best estimation for g in $\widetilde{t}_{\widetilde{h}} = \widetilde{h}g + \widetilde{\epsilon}_{\widetilde{h}}$, but the equation we have is of the form $t_h = hg + l$. Put $\mathbf{h} = (h_0, h_0 + 1, ..., h_1 - 1, h_1)^T$. Since

$$\mathbb{E}\mathbf{t} = \mathbb{E}\left(g\mathbf{h} + l + \boldsymbol{\epsilon}\right) = g\mathbb{E}\mathbf{h} + l + \mathbb{E}\boldsymbol{\epsilon} = g\mathbb{E}\mathbf{h} + l + \mathbb{E}\boldsymbol{\epsilon},$$

we can now write $\widetilde{t}_{\widetilde{h}} = t_h - \mathbb{E}\mathbf{t} = hg + l + \epsilon_h - (g\mathbb{E}\mathbf{h} + l + \mathbb{E}\boldsymbol{\epsilon}) = \widetilde{h}g + \widetilde{\epsilon}_h$, with $\widetilde{h} = h - \mathbb{E}\mathbf{h} = h - \frac{h_1 + h_0}{2}$ and $\widetilde{\epsilon}_{\widetilde{h}} = \epsilon_h$. We can further simplify the algorith by

$$\|\mathbf{h} - \mathbb{E}\mathbf{h}\|_2 = \sum_{h\in\mathbf{h}}(h - \mathbb{E}\mathbf{h})^2 = \frac{1}{12}(h_1 - h_0)(h_1 - h_0 + 1)(h_1 - h_0 - 2).$$

We now have the estimation for g, but not yet for l. We now have the approximation $\hat{t}(h) = h \cdot g + l$. We have translated $(\mathbb{E}\mathbf{h}, \mathbb{E}\mathbf{t})$ to $(0, 0)$ so that it is a point on the linear approximation in the origin. Translating this back, we get that $(\mathbb{E}\mathbf{h}, \mathbb{E}\mathbf{t})$ must be on the linear approximation, that is $\hat{t}(\mathbb{E}\mathbf{h}) = \mathbb{E}\mathbf{h} \cdot g + l = \mathbb{E}\mathbf{t}$. We can compute the value of l from this equality by $l = \mathbb{E}\mathbf{t} - \mathbb{E}\mathbf{h} \cdot g$. We now have all the ingredients fo the algorithm. The least squares algorithm adapted for h-relations is shown in Algorithm 5.1. The assignment of $\mathbf{h}$ is only symbolic. The difference $(h - \overline{h})$ and index in the computation of g can be done in the loop iteration.

What remains now is the measurement of t_h. This can simply be done by measuring the time of each h-relation several times and taking the average to get a more stable approximation. The measurement is done on full h-relations, to give a worst case estimation of the time per h-relation. Theoretically, there should be no difference in the time of an h-relation and a full h-relation, but in practice there is often a slight difference. This is done in the `bspbench` of the `BSPedupack`. Included with the library is a modernized version, which uses more of the `C++` data structures. The modernized version also contains an

optimized measurement of the number of flops per second, where the computation loop is written in assembly to reduce the effects of loop overhead. This assembly version is also more stable when compiled on different compilers, which optimize loops differently if written in `C++`.

Algorithm 5.1: Ordinary Least Squares for the `bspbench`

1 $[g, l]$:= **function** `OLS` $(h_0, h_1, \boldsymbol{t})$
2 $\quad \mathbf{h} := (h_0, h_0 + 1, ...h_1 - 1, h_1)^T$;
3 $\quad \bar{t} := \frac{\sum_{h \in \mathbf{h}} t_h}{h_1 - h_0 + 1}$;
4 $\quad \bar{h} := \frac{h_1 + h_0}{2}$;
5 $\quad g := \left(\sum_{h \in \mathbf{h}} (h - \bar{h}) \cdot (t_h - \bar{t})\right) \cdot \frac{12}{(h_1 - h_0)(h_1 - h_0 + 1)(h_1 - h_0 - 2)}$;
6 $\quad l := \bar{t} - \bar{h} \cdot g$;
7 **end function**

Now that we have the theory on how to benchmark the library, this can be brought into practice. The `BSPedupack` contained a benchmark for a general BSP implementation. The benchmark in `BSPedupack` is aimed at `Put` communication. As discussed in chapter 2, the speed `Put`, `Get` and `Send` is probably not equal. To see how this turns out in practice, Zefiros-BSPLib has a modernized version of this benchmark tool, containing benchmarks for all three types of communication. `BSPedupack` contains two versions of the least squares algorithm. One of the versions is quite similar to the least squares algorithm explained above, but differs in the way that the algorithm is adapted to the form of the data. A different approach is used to solve the problem that the data does not fit a straight line through the origin, but rather a straight line intersecting the y-axis somewhere above the origin.

The benchmark results of Zefiros-BSPLib are shown in Figure 5.1. We can see that indeed, `Put` is the fastest of the three. The penalty of using `Get` is not that big, but shows an overhead due to the use of two requests, `Get` requests and Buffered `Get` requests. `Send` is far more expensive due to the merging of the communication queues. This shows that our expectations were right. Improvements could be made on `Send` if we could figure out a way to prevent the extra copy of memory induced by the merging of the queues.

The implementation before this thesis was not yet faster than the MulticoreBSP implementation, mostly on par, in terms of time. To see if the improvements in this thesis really made a difference compared to the MulticoreBSP implementation, a comparison has to be made with the benchmark applied to MulticoreBSP . Due to the difference in compilers, `C` for MulticoreBSP and `C++` for Zefiros-BSPLib, the computation rate shown in the benchmark is quite different.

The experiments with `bspbench` are conducted on a desktop computer with a four core hyper-threaded processor. This is a virtualization technique to simulate the existence of eight cores, with latency hiding. Up to four cores can be used optimally. `bspbench` applied to MulticoreBSP measures a computation rate of approximately 475 Mflops/s, whereas Zefiros-BSPLib measures a computation rate of approximately 7000 Mflops/s on the same machine. When optimized using the assembly loop, the computation rate of Zefiros-BSPLib is approximately 10000 Mflops/s, while MulticoreBSP is only 6000 Mflops/s with the ported assembly loop. This shows how much this measure can differ on the same machine. The library is not to blame for this difference,

as this loop does not interact with the library. The difference is due to different ways of optimization by the different compilers. A more reliable comparison between the two libraries is to look at the last h-relation of both. For `Put`, this is 0.030408 ms versus 0.012462 ms for MulticoreBSP versus Zefiros-BSPLib.

In order to fully compare the two libraries, we need to measure each type of communication. `bspbench` from `BSPedupack` can easily be adapted to measure a different type of communication. We see that Zefiros-BSPLib beats MulticoreBSP on every type of communication. We can also see that the ordering in terms of speed is roughly the same. `Get` and `Put` are comparable in terms of speed, with `Put` having a slight advantage over `Get`. `Send` is much slower for both libraries.

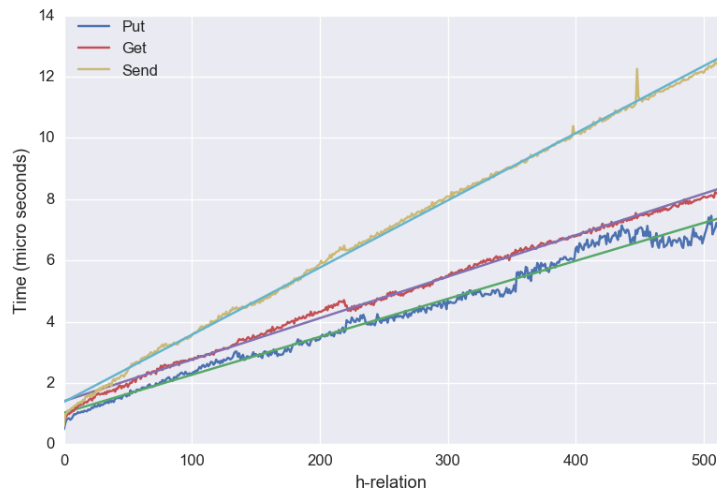


FIGURE 5.1: Plot of the timings of the different h-relations, fitted with the least squares approximation. Timings are in μs.

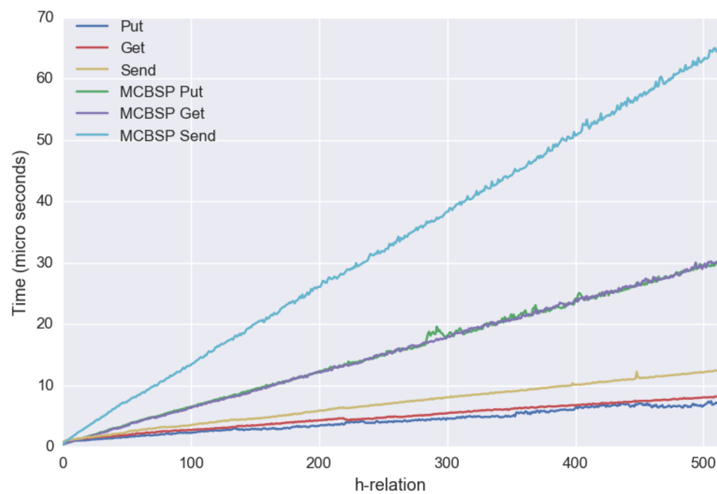


FIGURE 5.2: Plot of the timings of Zefiros-BSPLib versus MulticoreBSP . Timings are in μs.

5.2 LU decomposition

The second example is the LU decomposition in the `BSPedupack`. This section will be a summary of the LU decomposition in [1], with the addition of some interesting profiler images. The LU decomposition in `BSPedupack` uses a slightly different distribution of the data, which we will discuss briefly without going too much into detail. Since we are working with matrices here, it is more natural to think of the processors as a matrix of processors. This changes the enumeration of processors to $P(s,t)$, with $0 \leq s < M$ and $0 \leq t < N$, where M is the number of processor rows and N is the number of processor columns. The distribution used in the LU decomposition is a **matrix distribution**. The distribution function of a matrix distribution is of the form

$$\phi : \{(i,j) : 0 \leq i,j < n\} \rightarrow \{(s,t) : 0 \leq s < M \wedge 0 \leq t < N\},$$

so it maps an element of the data matrix to a processor in the processor matrix. It can be written as

$$\phi(i,j) = (\phi_0(i,j), \phi_1(i,j)).$$

Moreover, if it can also be written as

$$\phi(i,j) = (\phi_0(i), \phi_1(j)),$$

that is ϕ_0 is independent of j and ϕ_1 independent of i, the matrix distribution is called **Cartesian**. The LU decomposition uses the $M \times N$ cyclic matrix distribution for its matrix. This is described by

$$\phi(i,j) = (i \mod M, j \mod N).$$

Before we can parallelize the algorithm, we first need to know the basics about an LU decomposition, and the sequential algorithm we will be parallelizing. The LU decomposition decomposes an $n \times n$ non-singular matrix A into a lower triangular part L and an upper triangular part U, both $n \times n$ matrices, such that $LU = A$. This is useful for solving linear systems like $Ax = b$. Due to the structure of triangular matrices, solving becomes easy. It can be done row by row, starting from the bottom for upper triangular matrices, and from the top for lower triangular matrices. Each row immediately gives a solution for one new variable. We begin by solving $Ly = b$ for y, and then solving $Ux = y$ for x. The solution then satisfies $Ax = (LU)x = L(Ux) = Ly = b$. This is one example of the uses of an LU decomposition.

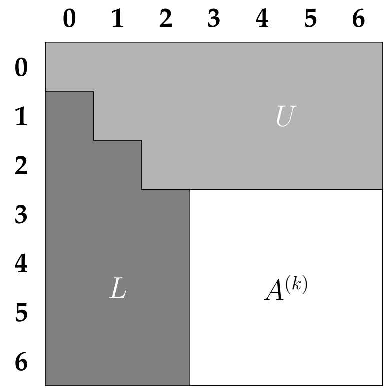


FIGURE 5.3: LU decomposition of a 7×7 matrix at the start of stage $k = 3$. The values of L and U computed so far and the computed part of $A^{(k)}$ fit exactly in one matrix[1].

The algorithm most suitable for parallelization, that is also more robust, is the LU decomposition with partial pivoting. During the algorithm, a permutation matrix P, which is just an identity matrix with swapped rows, will also be computed, such that $LU = PA$. The permutation matrix P comes from the partial pivoting during the algorithm. Every iteration, a pivot row will be chosen, by looking at the nonzero column below and choosing the one with the largest absolute value. This is done to ensure we are not eliminating with a zero pivot element, as this would cause division by zero. This row will be swapped with the top row of the remaining sub-matrix, and will be used to eliminate the first non-zero column below. The pivot row will be stored in U, and the factors of the elements in the column below the pivot row, divided by the pivot element, will be stored in the next column of L. Because the pivot element divided by itself is always 1 (provided the matrix is non-singular), the diagonal of the matrix L will be the identity matrix. Thus, $L - I_n$ will be strictly lower diagonal, that is, it has only 0 on its diagonal. Every iteration, one row of U will become fixed, and one column of L will become fixed. Since the rows of U will be 0 where the column of $L - I_n$ is nonzero, and vice versa, we can store the values of the two matrices in the same matrix. The nonzero elements do not overlap. The remaining sub-matrix in the lower right corner exactly fits the remaining part of A. This is more memory efficient, since we only need to allocate one matrix instead of three. It is visualized in Figure 5.3. In the algorithm the matrix A will be used as the matrix of the current state, containing the fixed parts of L and U, and the remaining sub-matrix which is yet to be processed. The input matrix will be called $A^{(0)}$ and the output matrix will be A in its final state, such that $A = (L - I_n) + U$.

We can extract L by taking the lower triangle of A, excluding the diagonal, and adding I_n. The upper triangle, including the diagonal, will be U. We now have the equality $LU = PA^{(0)}$, or equivalently $LUx = PA^{(0)}x = Pb$. Suppose we now want to solve $A^{(0)}x = b$ again. We now begin by solving $Ly = Pb$ for y, and then $Ux = y$ for x. The solution then satisfies

$$A^{(0)}x = (P^{-1}P)A^{(0)}x = P^{-1}(PA^{(0)})x = P^{-1}LUx == P^{-1}Ly = P^{-1}Pb = b,$$

as desired. Finally, the permutation matrix can also be stored as a vector π of

[1] This figure is reconstructed from [1].

length n to reduce memory usage, and for simpler permutations in the program. Row swapping using the permutation vector is much cheaper than matrix-matrix multiplication with a permuted identity matrix. The sequential algorithm is shown in Algorithm 5.2. The sequential cost can be easily determined by translating the for loops to summations of the number of flops in the loop body. For the sake of simplicity, we will mainly focus on floating point operations, not on swaps, assignments or comparisons for determining the maximum element in the column. In the real algorithm, they do contribute, but not as dominantly as the floating point operations. Starting with the inner loops, we can easily see that for a certain k in the outer loop, the cost of the double loop over i and j is $(n-k-1)(n-k-1)$ iterations, times one division and one subtractions, so $2(n-k-1)^2$ flops. The loop before that has another $n-k-1$ divisions. We now need to sum over k for the outer loop, so the final cost will be

$$T_{seq} = \sum_{k=0}^{n-1} \left(2(n-k-1)^2 + n-k-1\right) = \sum_{k=0}^{n-1}(2k^2+k) = \frac{2n^3}{3} - \frac{n^2}{2} - \frac{n}{6},$$

which can easily be proven by induction over n.

Algorithm 5.2: Sequential LU decomposition with partial row pivoting[2].

Input : A: $n \times n$ matrix, $A = A^{(0)}$.
Output: A: $n \times n$ matrix, $A = L - I_n + U$, with
L: $n \times n$ unit lower triangular matrix,
U: $n \times n$ upper triangular matrix,
π: permutation vector of length n,
such that $a^{(0)}_{\pi(i),j} = (LU)_{ij}$, for $0 \le i, j < n$.

for $i := 0$ **to** $n-1$ **do**
  $\pi_i := i$;
for $k := 0$ **to** $n-1$ **do**
  $r := \arg\max(|a_{ik}| : k \le i < n)$;
  swap(π_k, π_r);
  for $j := 0$ **to** $n-1$ **do**
    swap(a_{kj}, a_{rj});
  for $i := k+1$ **to** $n-1$ **do**
    $a_{ik} := a_{ik}/a_{kk}$;
  for $i := k+1$ **to** $n-1$ **do**
    for $j := k+1$ **to** $n-1$ **do**
      $a_{ij} := a_{ij} - a_{ik}a_{kj}$;

In order to parallelize the algorithm, we need to distribute the input and output data. The matrix will be $M \times N$ cyclic, as discussed. The permutation vector π will be distributed the same way as the first column of the matrix is distributed, and only processor column 0 will store this information, as it is not really needed during computations, only for output. The pivot element is computed in parallel by the processors of the first column of the remaining part of A. Each processor determines the index of element with the largest absolute value in their local memory, then communicates both the index and the value of the largest element to the other processors in the same column.

[2]This algorithm is reconstructed from [1].

Then all processors in that column determine the index of the global maximum element, and they also compute the values that have to be stored in the next column of L. Then they communicate the index of the pivot row to all the processors in the same processor row. This is done in supersteps $(0) - (3)$ of Algorithm 5.3. In supersteps $(4) - (5)$, the row swap is performed. It has to be done in two supersteps, because the two rows are (usually) not stored in the same processors. The variables with ˆ are temporary variables. In superstep (0′), each processor in row $i \mod M$ needs to know the factor previously stored in a_{ik} from column k, which is now part of L. Only processor $P(i \mod M, k \mod N)$ knows the value, so we need to somehow broadcast it to the entire row. The same goes for the value a_{kj} from row k, which is the pivot row that is now part of U. Each processor in column $j \mod N$ needs to know this value, so we need to broadcast it to the entire column somehow. For this purpose, a two-phase broadcast is used, which is why the broadcast has two superstep numbers assigned. Phase 0 of each broadcast is merged together into one superstep, superstep (6), and phase 1 of each broadcast is done in superstep (7).

The two-phase broadcast is discussed and explained in much detail in [1], but will only briefly and textually be explained here. Broadcasting a vector of length n from one processor to all other processors would have a BSP cost of $(p-1)ng + l$. The two-phase broadcast first distributes the vector over all processors. The processor keeps $\left\lceil \frac{n}{p} \right\rceil$ elements to itself, and also sends that amount to each of the other processors, so that each processor has approximately the same number of elements of that vector. This is phase 0 of the two-phase broadcast, which has a BSP cost of $(n - \left\lceil \frac{n}{p} \right\rceil)g + l$. In phase 1, each processor sends his part of the vector to all other processors. This phase has a BSP cost of $(p-1)\left\lceil \frac{n}{p} \right\rceil + l$. The total BSP cost for the two phase broadcast will be

$$T_{\text{broadcast}} = (n + (p-2)\left\lceil \frac{n}{p} \right\rceil)g + 2l \approx 2ng + 2l,$$

which is much less than $(p-1)ng + l$, as long as of course $p > 3$ and l is not too large.

For a $M \times N$ cyclic distribution, we need $p = M \cdot N$ available processors. The complete cost analysis in [1], but the important part is that $M \approx N \approx \sqrt{p}$ is the optimal choice for M and N, and the BSP cost of the algorithm is approximately

$$T_{LU} \approx \frac{2n^3}{3p} + \frac{3n^2}{2\sqrt{p}} + \frac{3n^2 g}{\sqrt{p}} + 8nl,$$

for this choice of M, N. The dominating factor in the sequential algorithm is $\frac{2n^3}{3}$. The dominating factor in the parallel algorithm is now $\frac{2n^3}{3p}$, which is a factor p smaller than the sequential algorithm. Of course, the entire cost is still not reduced by a factor p, but at least it will come close. This is always what we are after. The LU decomposition is an interesting test case, because we can see how much the improvements made to the library affect the performance of a computation bound algorithm.

The `BSPedupack` contains a `bsplu` function and an executable program to test the function, `bsplu_test`. These can be compiled without changes to the code using Zefiros-BSPLib. However, to get useful profiler output, we need to add some annotations in the program. The version with annotations

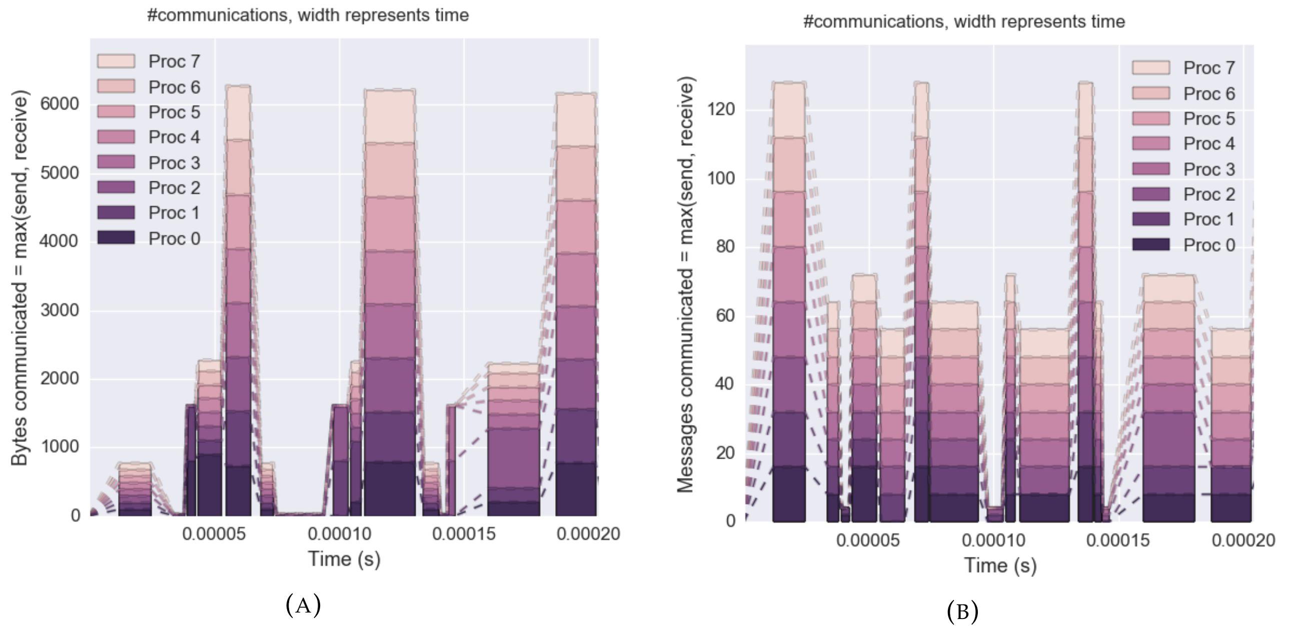
is in Appendix A. The textual supersteps do not have a one on one correspondence with the program supersteps. As mentioned before, the consecutive computation and communication supersteps can often be merged into only one program superstep. As a result, the program merges supersteps {(0′), (0), (1)} into `Superstep 0`, {(2), (3)} into `Superstep 1`, {(4),(5)} into `Superstep 2`, and renames (6) and (7) to `Superstep 3` and `Superstep 4`. The merge of {(4),(5)} might seem odd, because we first have communication, and then assignment of temporary variables into the real matrix, but this is only to make it textually more clear what is happening. The temporary variables are induced by the communication buffers, and are only used textually to prevent confusion about the state of the variables.

The LU decomposition has some interesting profiler images. Figure 5.4 shows the number of bytes communicated and the number of requests that were used to communicate those bytes. The width of the bars represents the time it took to communicate and to synchronize. The space between the bars is computation time. Figure 5.4 (A) is similar to the image in [1] that was created using Oxford BSP toolset profiler [4]. This new image is created using the Zefiros-BSPLib Profiler. Figure 5.4 (B) gives some more insight into why some of the rather tiny bars in (A) have such a large width. It shows that both the size in bytes and the number of requests affect the communication and synchronization time. The latter is more likely to be of influence on shared memory architectures. On distributed memory architectures, all information is bundled before it is sent to another processor, and the actual sending and receiving of the bundled information will be much more expensive than the separation back into requests. On shared memory, both have a large influence on the communication time. Figure 5.5 is quite similar, but now it just shows the average number of bytes and the average number of requests in each superstep. The vertical lines are confidence intervals for the communication amount in that superstep. It is now more clear that the amount of bytes and the amount of requests does not change in `Superstep 0` and `Superstep 1`, but it does change in the other supersteps. This is due to the decrease in size of the remaining part of A. Figure 5.6 shows the amount of time each processor takes for computation and communication+synchronization in each superstep. It becomes more clear now that communication and synchronization takes a huge portion of the time. Time varies per processor, but not too much. Figure 5.7 shows the patterns in communication more clearly. We can now also see that the pattern does not change too much over the course of the algorithm, the amount of bytes just decreases. This can be seen by the fading colors in the supersteps. Figure 5.8 shows the ratio between computation and communication+synchronization per superstep. Due to the small timing values, this sometimes leads to a ratio 0, which is never true. What we can see from this image, is that most of the time, the ratio varies around $\frac{1}{2}$. This indicates that the algorithm is communication bound, or at least for this matrix size and distribution.

It would now be interesting to compare this example between MulticoreBSP and Zefiros-BSPLib. As it turns out, the difference is not too large. Both libraries are tested on a 200×200 matrix, with $M = 4$ and $N = 1$. The large synchronization time we saw in the profiler output could partly be due to data collection by the profiler. This should be filtered out for the profiler output in a later release. The timing is 0.001234 seconds versus 0.001304 seconds for Zefiros-BSPLib versus MulticoreBSP . There is a slight advantage of Zefiros-BSPLib over MulticoreBSP , and as we noticed in the cost analysis, the LU

decomposition is mainly computation bound, and the difference is indeed not that large between libraries, which usually indicates that most of the time is spent on computation.

This observation differs from the profiler output in Figure 5.6. This profiler output showed that communication took a large part of the time. This is probably explained by Figure 5.4, where we can see that the number of requests is usually large. A large numbers of requests causes extra overhead over a small number of requests with the same size, and even more so with the profiler enabled.



(A) (B)

FIGURE 5.4: Profiler output showing the number of bytes and the number of requests communicated for the parallel LU decomposition with $M = 8, N = 1, n = 100$. Width of the bars and the gaps represent communication and computation time respectively. The zoom functionality of the GUI is used to zoom in on the initial stages of the algorithm.

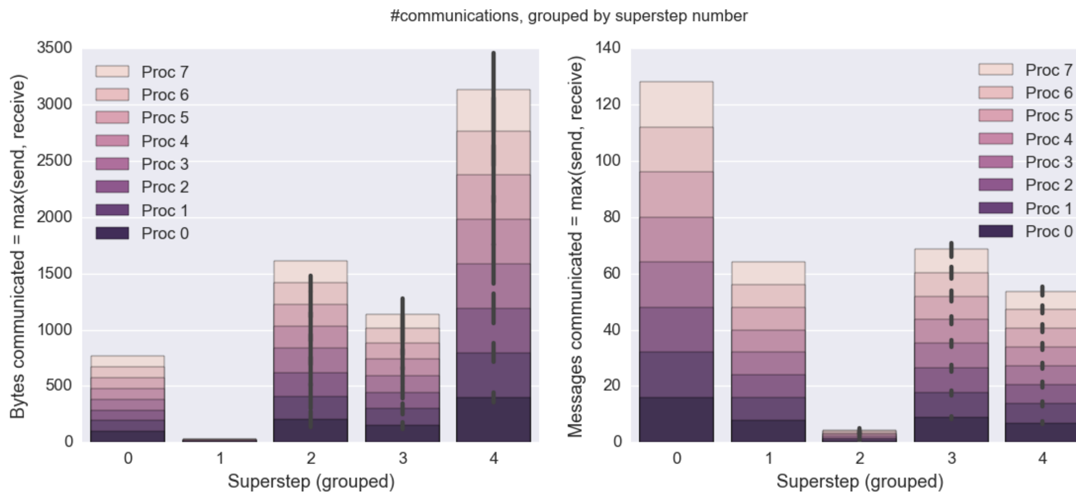


FIGURE 5.5: Profiler output showing the number of bytes and the number of requests communicated for the parallel LU decomposition with $M = 8, N = 1, n = 100$. The bars are grouped by superstep. The vertical lines represent confidence intervals in which the amount varies for that superstep.

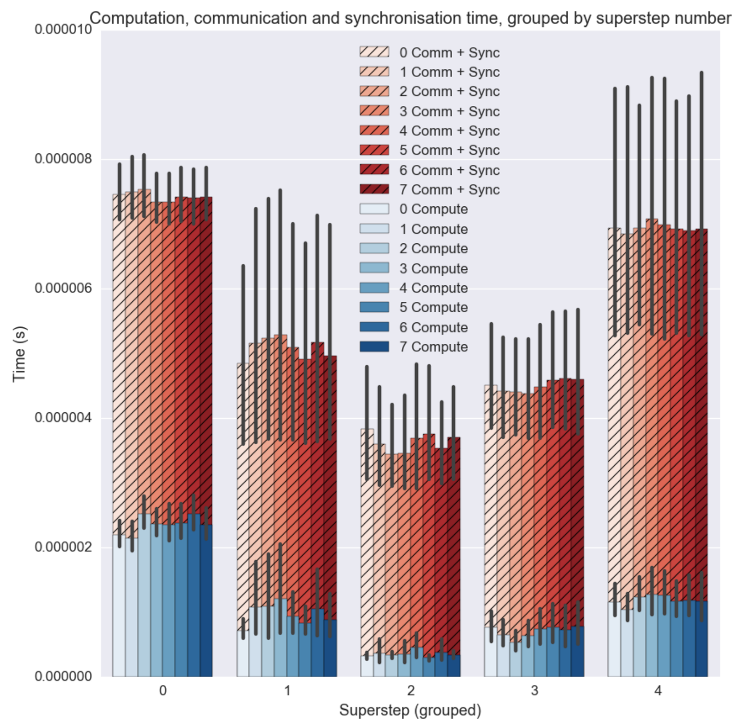


FIGURE 5.6: Profiler output for the parallel LU decomposition with $M = 8, N = 1, n = 100$, showing the time each processor takes in each superstep. The time is separated into computation time and computation and synchronization time. The vertical lines are confidence intervals for the times.

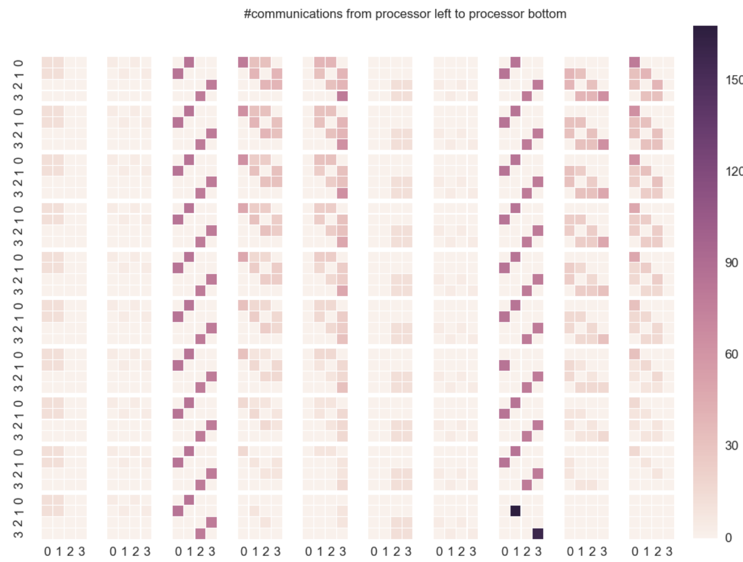


FIGURE 5.7: Profiler output for the parallel LU decomposition with $M = 2, N = 2, n = 20$, showing the communication pattern per superstep in each small matrix. The supersteps are read from left to right, row by row.

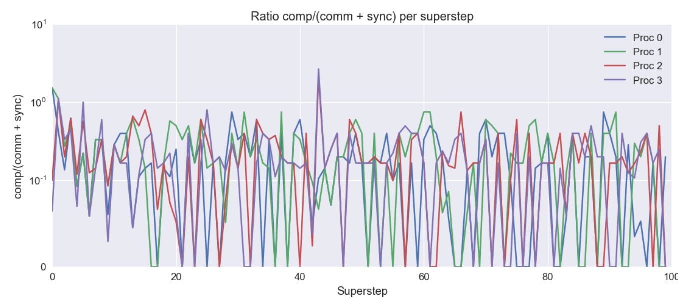


FIGURE 5.8: Profiler output for the parallel LU decomposition with $M = 2, N = 2, n = 20$, showing ratio between computation and communication+synchronization per superstep.

Algorithm 5.3: Parallel LU decomposition algorithm for $P(s,t)$.[3]

Input : A: $n \times n$ matrix, $A = A^{(0)}$, distr(A) $= M \times N$ cyclic.
Output: A: $n \times n$ matrix, distr(A) $= M \times N$ cyclic, $A = L - I_n + U$, with
L: $n \times n$ unit lower triangular matrix,
U: $n \times n$ upper triangular matrix,
π: permutation vector of length n, distr(π) = cyclic in $P(*,0)$,
such that $a^{(0)}_{\pi(i),j} = (LU)_{ij}$, for $0 \le i,j < n$.

```
     if t = 0 then for all i : 0 ≤ i < n ∧ i mod M = s do
         π_i = i;
     for k := 0 to n − 1 do
         if k mod N = t then
(0)          r_s := arg max(|a_ik| : k ≤ i < n ∧ i mod M = s);
(1)          put r_s and a_{r_s,k} in P(∗, t);
(2)          s_max := arg max(|a_{r_q,k}| : 0 ≤ q < M);
             r := r_{s_max};
             for all i : k ≤ i < n ∧ i mod M = s ∧ i ≠ r do
                 a_ik := a_ik / a_rk;
(3)          put r in P(s, ∗);

(4)      if k mod M = s then
             if t = 0 then put π_k as π̂_k in P(r mod M, 0);
             for all j : 0 ≤ j < n ∧ j mod N = t do
                 put a_kj as â_kj in P(r mod M, t);
         if r mod M = s then
             if t = 0 then put π_r as π̂_r in P(k mod M, 0);
             for all j : 0 ≤ j < n ∧ j mod N = t do
                 put a_rj as â_rj in P(k mod M, t);

(5)      if k mod M = s then
             if t = 0 then π_k := π̂_r;
             for all j : 0 ≤ j < n ∧ j mod N = t do
                 a_kj := â_rj;
         if r mod M = s then
             if t = 0 then π_r := π̂_k;
             for all j : 0 ≤ j < n ∧ j mod N = t do
                 a_rj := â_kj;

(6)/(7)  broadcast((a_ik : k < i < n ∧ i mod M = s), P(s, k mod N), P(s, ∗));
(6)/(7)  broadcast((a_kj : k < j < n ∧ j mod N = t), P(k mod M, t), P(∗, t));

(0′)     for all i : k < i < n ∧ i mod M = s do
             for all j : k < j < n ∧ j mod N = t do
                 a_ij := a_ij − a_ik a_kj;
```

[3]This algorithm is reconstructed from [1].

5.3 Fast Fourier Transform

Finally, we will discuss the **F**ast **F**ourrier **T**ransform (FFT) that is included in `BSPedupack`. This section is again a summary of the beginning of a chapter in [1], stating some of the points what makes the FFT a suitable and interesting test case. The FFT is a faster algorithm for the **D**iscrete **F**ourier **T**ransform (DFT). The DFT is a way to compute the Fourier Transform of a discrete vector. A Fourier Transform is a way to express a ***T*-periodic** function $f : R \rightarrow C$ as a series of complex powers of e. A T-periodic function is a function that satisfies $f(t+T) = f(t)$ for all $t \in R$. The Fourier series associated with f is of the form

$$\tilde{f}(t) = \sum_{k=-\infty}^{\infty} c_k \mathrm{e}^{2\pi ikt/T},$$

where the Fourier coefficients are defined by

$$c_k = \frac{1}{T} \int_0^T f(t)\mathrm{e}^{-2\pi ikt/T} dt.$$

To discretize this, we could for example sample n equidistant points of the interval $[0, T]$. Then using the trapezoidal rule for numerical integration, we obtain

$$\begin{aligned} c_k &= \frac{1}{T} \int_0^T f(t)\mathrm{e}^{-2\pi ikt/T} dt \\ &\approx \frac{1}{T} \cdot \frac{T}{n} \left(\frac{f(0)}{2} + \sum_{j=1}^{n-1} f(t_j)\mathrm{e}^{-2\pi ikt_j/T} + \frac{f(T)}{2} \right) \\ &= \frac{1}{n} \sum_{j=0}^{n-1} f(t_j)\mathrm{e}^{-2\pi ijk/n}. \end{aligned}$$

The DFT of a complex vector $\mathbf{x} = (x_0, x_1, ..., x_{n-1})^T \in \mathbb{C}^n$ is then defined by a vector $\mathbf{y} = (y_0, y_1, ..., y_{n-1})^T \in \mathbb{C}^n$ with

$$y_k = \sum_{j=0}^{n-1} x_j \mathrm{e}^{-2\pi ijk/n}, \quad \text{for } 0 \leq k < n.$$

The inverse of a DFT is simply defined by

$$x_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \mathrm{e}^{2ijk/n}, \quad \text{for } 0 \leq j < n.$$

This is equivalent to the DFT itself, except for the sign in the power of e and the factor $\frac{1}{n}$. This would require $n - 1$ complex additions and n complex multiplications if we assume the powers of e have been precomputed and stored in a table. Complex addition requires two real additions, since $(a + bi) + (c + di) = (a + c) + (b + d)i$, whereas complex multiplication requires one real addition, one real subtraction and four real multiplications, since $(a + bi)(c + di) = (ac - bd) + (ad + bc)i$. There are n entries in $\mathbf{y}$, all requiring $8n - 2$ flops. This results in a sequential cost of $8n^2 - 2n$ flops. We use $\omega_n = \mathrm{e}^{-2\pi i/n}$, so that we can write $\mathrm{e}^{-2\pi ijk/n} = \omega_n^{jk}$. In matrix form, this

becomes $\mathbf{y} = F_n\mathbf{x}$, with $(F_n)_{jk} = \omega_n^{jk}$.

The FFT algorithm improves on this in a very simple and elegant way, by a (not as simple) observation. Since $\omega_n^2 = \mathrm{e}^{(-2\pi ijk/n)\cdot 2)} = \omega_{n/2}$, we can split the sum into even and odd indices and write

$$\begin{aligned} y_k = \sum_{j=0}^{n-1} x_j\omega_n^{jk} &= \sum_{j=0}^{n/2-1} x_{2j}\omega_n^{2jk} + \sum_{j=0}^{n/2-1} x_{2j+1}\omega_n^{(2j+1)k} \\ &= \sum_{j=0}^{n/2-1} x_{2j}\omega_{n/2}^{jk} + \omega_n^k \sum_{j=0}^{n/2-1} x_{2j+1}\omega_{n/2}^{jk}, \end{aligned}$$

assuming n is even. We can recognize two FFT in these sums, one for the odd and one for the even indices. This new Fourier transform is of length $n/2$, so we have to restrict the output indices k to $0 \leq k < n/2$. To get an expression for the output indices $n/2 \leq k < n$, we introduce $k' = k - n/2$, which now satisfies $0 \leq k' < n/2$ again. By observing $\omega_{n/2}^{n/2} = 1, \omega_n^{n/2}$, we get that

$$\begin{aligned} y_k &= y_{k'+n/2} \\ &= \sum_{j=0}^{n/2-1} x_{2j}\omega_{n/2}^{j(k'+n/2)} + \omega_n^{k'+n/2} \sum_{j=0}^{n/2-1} x_{2j+1}\omega_{n/2}^{j(k'+n/2)} \text{ for } 0 \leq k' < n/2 \\ &= \sum_{j=0}^{n/2-1} x_{2j}\omega_{n/2}^{jk'} - \omega_n^{k} \sum_{j=0}^{n/2-1} x_{2j+1}\omega_{n/2}^{jk'} \text{ for } 0 \leq k' < n/2. \end{aligned}$$

The sums are again in the form of an FFT, but now for $n/2 \leq k < n$. Now comes the elegant part: if we look at the sums, we see that they the first sum in both equations is equal, and that the second sum is also equal in both equations. The only difference is that we add them up in the first equation, and subtract them in the second equation. We now only have to compute both sums once, and if we add them up, we get y_k, if we subtract the second from the first, we get $y_{k+n/2}$. If we first compute all the half length Fourier transforms with the DFT algorithm, this would cost $2\cdot\big(8(n/2)^2 - 2(n/2)\big) = 4n^2 - 2n$ flops. Combining the results would cost $n/2$ complex multiplications, for the second sum with the coefficient ω_n^k in front, $n/2$ complex additions for elements $0 \leq k < n/2$ and $n/2$ subtractions for elements $n/2 \leq k < n$, resulting in $(6 + 2 + 2) \cdot (n/2)$ flops. The total cost is already reduced from $8n^2 - 2n$ to $4n^2 + 3n$ flops. Instead of using the DFT, we could also apply FFT for the smaller sums. This requires $n/2$ to also be even. If we want to repeat this until we are only left with a single element in the sum, we would require n to be a power of 2, which we will assume from now on for simplicity.

This algorithm is recursive, because each FFT applies two FFT computations for its own sums. To analyze the cost of this recursive algorithm, we can express the cost function of an FFT of length n recursively by

$$T(n) = 2T\left(\frac{n}{2}\right) + 5n.$$

Following the recursion, we obtain

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + 5n = 2\left(2T\left(\frac{n}{4}\right) + 5\frac{n}{2}\right) + 5n \\ &= 4T\left(\frac{n}{4}\right) + 2 \cdot 5n = ... = nT(1) + \log_2 n \cdot 5n = 5n \log_2 n, \end{aligned}$$

which is already far less than $8n^2 - 2n$.

As recursive algorithms are hard to parallelize, we would like to formulate it as a non-recursive algorithm first. In order to do this, we first rewrite the recursive computation in matrix language. It can be written as

$$F_n\mathbf{x} = \begin{bmatrix} I_{n/2} & \Omega_{n/2} \\ I_{n/2} & -\Omega_{n/2} \end{bmatrix} \begin{bmatrix} F_{n/2} & 0 \\ 0 & F_{n/2} \end{bmatrix} \begin{bmatrix} x(0:2:n-1) \\ x(1:2:n-1) \end{bmatrix}, \tag{5.1}$$

where $\Omega_n = \text{diag}(1, \omega_{2n}, \omega_{2n}^2, ..., \omega_{2n}^{n-1})$. Note that the vector $[x(0 : 2 : n - 1), x(1 : 2 : n-1)]$ is a permutation of the vector **x**, which can also be achieved using a permutation matrix. We will denote this by S_n, the **even-odd sort matrix**, such that

$$S_n\mathbf{x} = \begin{bmatrix} x(0:2:n-1) \\ x(1:2:n-1) \end{bmatrix}. \tag{5.2}$$

Each of the matrices $I_{n/2}, \Omega_{n/2}, F_{n/2}$ are $n/2 \times n/2$ matrices, forming $n \times n$ matrices by concatenation. The matrix containing two identical copies of $F_{n/2}$ is a block diagonal matrix, with all zero entries on the off-diagonal blocks. These blocks can also be interpreted as $0 \cdot F_{n/2}$. This is a useful observation, as we can now simplify the notation using the **Kronecker product** of matrices $A \otimes B$. Let A be a $q \times r$ matrix and B a $m \times n$ matrix. Then $A \otimes B$ is a $qm \times rn$ matrix defined by

$$A \otimes B = \begin{bmatrix} a_{00}B & \dots a_{0,r-1} & B \\ \vdots & & \vdots \\ a_{q-1,0}B & \dots & a_{q-1,r-1}B \end{bmatrix}.$$

Among the many useful properties of the Kronecker product, there are three we will now be using.

Lemma 5.3.1. The Kronecker product has the following properties.

i. Let A, B, C be matrices. Then

$$(A \otimes B) \otimes C = A \otimes (B \otimes C).$$

ii. Let A, B, C, D be matrices such that AC and BD are defined. Then

$$(A \otimes B)(C \otimes D) = AC \otimes BD.$$

iii. Let $m, n \in \mathbb{N}$. Then

$$I_m \otimes I_n = I_{mn}.$$

The middle matrix in (5.1) can now be written as $I_2 \otimes F_{n/2}$. The left most matrix will be labeled as B_n. The new notation is now $F_n\mathbf{x} = B_n(I_n \otimes F_{n/2})S_n\mathbf{x}$.

This holds for all vectors **x**, so we can conclude that

$$F_n = B_n(I_2 \otimes F_{n/2})Sn$$

is a matrix factorization of F_n. If we write the middle term in the form $I_k \otimes F_{n/k}$, and apply the factorization again, we obtain

$$\begin{aligned} I_k \otimes F_{n/k} &= [I_k I_k I_k] \otimes [B_{n/k}(I_2 \otimes F_{n/(2k)})S_{n/k}] \\ &= (I_k \otimes B_{n/k})([I_k I_k] \otimes [(I_2 \otimes F_{n/(2k)})S_{n/k}]) \\ &= (I_k \otimes B_{n/k})(I_k \otimes I_2 \otimes F_{n/(2k)})(I_k \otimes S_{n/k}) \\ &= (I_k \otimes B_{n/k})(I_{2k} \otimes F_{n/(2k)})(I_k \otimes S_{n/k}). \end{aligned}$$

Since each of the $I_k \otimes S_{n/k}$ is a permutation matrix, the product of all these matrices is again a permutation matrix. We will use the notation

$$R_n = (I_{n/2} \otimes S_2)...(I_4 \otimes S_{n/4})(I_2 \otimes S_{n/2})(I_1 \otimes S_n).$$

By repeatedly applying the factorization on the middle term, and collecting the right hand side, we obtain the following theorem.

Theorem 5.3.2. (Cooley and Tukey [5] — DIT)
Let n be a power of two with $n \geq 2$. Then

$$F_n = (I_1 \otimes B_n)(I_2 \otimes B_{n/2})(I_4 \otimes B_{n/4})...(I_n/2 \otimes B_2)R_n,$$

with R_n as described before.

All this is still to rewrite the algorithm into a form that can be parallelized. In the form we have now, we can first apply the permutations from $\mathbf{x}' = R_n\mathbf{x}$ at the beginning, and then we can iteratively multiply by $(I_{n/k} \otimes B_k)$, starting with $k = 2$ and multiplying k by 2 in each iteration. This is the basis for the non-recursive algorithm: we have no recursive definition of F_n anymore. Still, there are many complications to overcome before we can parallelize the algorithm. One of these difficulties is the distribution of the coefficients ω_n. This should be such that every processor can access the needed coefficients, without needing to store or precompute the entire table itself. Another is the distribution of the two vectors itself, and the split into supersteps. All this is discussed in detail in [1], but since we are interested in how the improvements of the library impact the efficiency of the algorithm, we will now skip to the cost analysis of the algorithm.

As explained in [1], the cost of the parallel FFT is greatly simplified in the case $p \leq \sqrt{n}$. Since we work on shared memory, the number of processors is limited. Therefore, any size n easily satisfies this inequality. The BSP cost is then

$$T_{FFT,1<p\leq\sqrt{n}} = \frac{5n\log_2 n}{p} + 2\frac{n}{p}g + 3l.$$

Only three supersteps are needed if the proper distribution is chosen: one computation, followed by one communication, and finally one more computation. In the first superstep, a block distribution is chosen. During the communication superstep, the vector is redistributed into group-cyclic distribution. During the computation, each processor distributes its own part to all processor that will own that part in the next superstep. This is done using a buffer, and the own elements are relocated as well, causing the processor to send and receive exactly $2\frac{n}{p}$ real numbers, twice the number of complex numbers. The

computation cost of the non-recursive algorithm would be the same as the recursive algorithm, as explained in [1]. The computations can be divided equally due to the assumptions on n, and we do not need to collect results afterwards anymore, so the resulting cost is just the cost of the sequential algorithm, divided by the number of processors, which is the optimal case for a parallel algorithm. What makes this an interesting test case, is the fact that the ratio between the amount of computations and the amount of communication is much smaller than for the LU decomposition.

To reduce the initialization time, some loops in the code were changed, for example $m = 1 : 1 : \log_2 n$, with $k = 2^m$ to replace a loop where k is initialized with value 2, and multiplied by 2 in every iteration. We can then multiply by k by shifting the bits of the binary representation of an integer m positions. If we label the bits by b_j, then each bit contributes to $b_j \cdot 2^j$ in the number, where $b_j \in \{0, 1\}$. Shifting the bits one position produces a number with $b'_{j+1} = b_j$, and $b_0 = 0$. Integer division can be done by shifting in the other direction. Binary operations such as those shifts are usually much cheaper than their multiplication or division counterparts. Multiplication or division by a constant power of two is often optimized by the compiler to be done with these binary shifts, but loop iterations can sometimes not be optimized in this way. To help the compiler optimize the loop, writing the binary shifts manually can increase the speed of the computation. This is not guaranteed, but it often does help.

In the initialization, this tactic does increase the speed of computations. However, for the FFT it can only be applied in one of the outer loops of the algorithm (for the algorithm, see [1]). This results in only a minor increase of speed, but the speed is more stable over different runs of the algorithm. Writing binary operations is often more obscuring than it is a speed gain. In cases like this, binary operations are justified, but should, as a rule of thumb, always be accompanied by a comment explaining the operation. For example:

```
1 // k = 2^m
2 // x = y / k
3 x = y >> m;
```

Looking at the computation and communication cost, the difference is not as huge as for the LU decomposition, a factor of $O(\log_2 n)$ instead of a factor of $O(n)$ for LU. We will now experiment to see how much we can improve by using Zefiros-BSPLib over MulticoreBSP . The experiment we will be looking at is a parallel FFT, using 4 processors and a complex vector of length 8192. In practice, the vector is often much larger, but we need to apply the FFT multiple times in order to get a stable average time per FFT. We do this by applying the FFT 10000 times, 5000 times FFT and 5000 times the inverse FFT. We also count initialization time, for the precomputation of the weights and the permutation matrix. This results in the following output of the algorithm in `BSPedupack`:

Zefiros-BSPLib

```
1 Time per initialization =
2                0.000115 sec
3 Time per FFT = 0.000048 sec
4 Computing rate in FFT =
5         11396.872477 Mflop/s
```

MulticoreBSP

```
1 Time per initialization =
2                0.000170 sec
3 Time per FFT = 0.000090 sec
4 Computing rate in FFT =
5         6119.505399 Mflop/s
```

Both initialization and FFT speed have increased significantly with Zefiros-BSPLib, showing that for algorithms with a large communication part, the cost can be greatly reduced by a faster library implementation.

The FFT also has some interesting profiler output. From Figure 5.10, it seems the initialization is imbalanced. However, each processor has a different parameter which is part of a $\sin$ and $\cos$ computation. The number of $\sin$ and $\cos$ operations is equal, and so is the rest of the initialization. This huge difference is due to the way the $\sin$ and $\cos$ are implemented. Several checks are done to determine in which range the number lies, and then different algorithms are used for different ranges. Apparently, the algorithm processor 0 had to use was much more costly. This is unpredictable over different architectures and different compilers. This program output is generated on Windows. Running the same program on Linux, with a different compiler, but the same computer, resulted in processor 0 and 4 being swapped in terms of computation time.

Figure 5.11 shows that each FFT computation takes roughly the same amount of time in terms of both computation and communication. However, the first FFT is slightly more costly, as buffers are allocated in the first FFT, and reused in the next. However, it is unusual to run a program and compute a single FFT. This is usually part of a larger algorithm, or in a series of more comparable FFT computations. The overhead of Zefiros-BSPLib is then only visible for the first or first few supersteps.

Figure 5.12 is not rather interesting to look at, as there are no special communication patterns like the LU decomposition had. However, this is in general a picture of an algorithm utilizing full h-relations.

Figure 5.9 shows that the ratio between computation and communication is rather stable over the different FFT computations, slightly more than a factor 10. Analyzing the cost of the parallel algorithm for these parameters, neglecting the synchronization cost for now, as we only need one synchronization, the ratio is approximately $\frac{5n\log_2 n}{p}/(\frac{2n}{p}g) = \frac{32\frac{1}{2}}{g}$. This would imply $g \approx 3$, but this is not really realistic, even on shared memory. The communication of the FFT is rather optimal, as we only need to separate the communication into p parts for this setup. This is not the general case. What it does show however, is that the BSP cost analysis is a good approximation of the cost of the parallel algorithm: the ratio between computation and communication matches the reality up to a minor factor, which is introduced by the fact that we neglect some of the code in our cost analysis, like loop iteration and allocation of variables in memory.

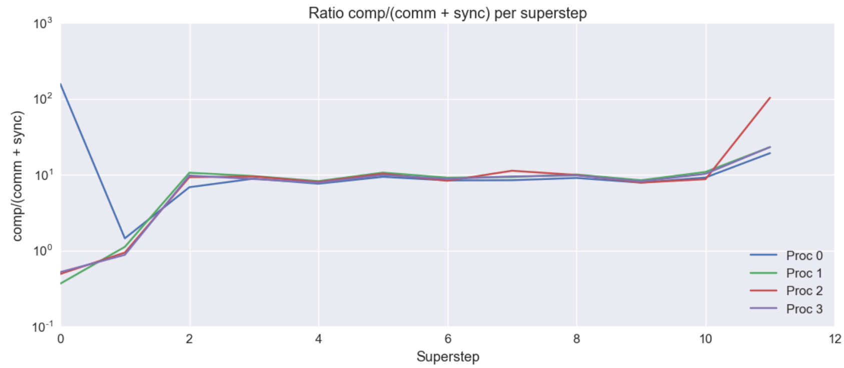


FIGURE 5.9: Profiler output for the FFT with $p = 4$ and $n = 8192$. The ratio per superstep is shown. The ratio seems to be stable for the FFT and inverse FFT, and slightly different for the initialization and de-initialization of the algorithm.

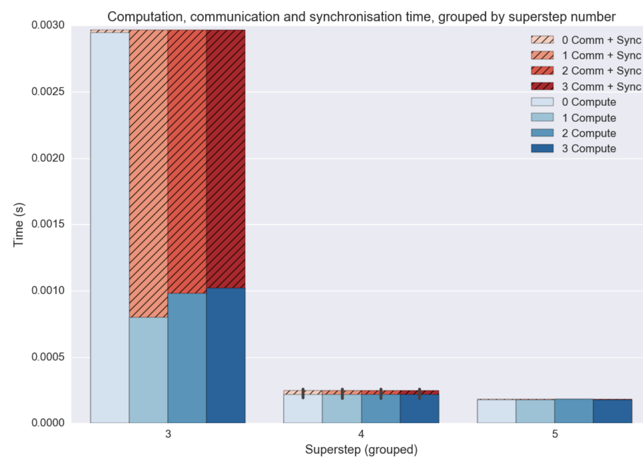


FIGURE 5.10: Profiler output for the FFT with $p = 4$ and $n = 8192$. Average computation, communication and synchronization time per superstep, grouped by superstep number. Averaged over 5 normal and 5 inverse FFT computations.

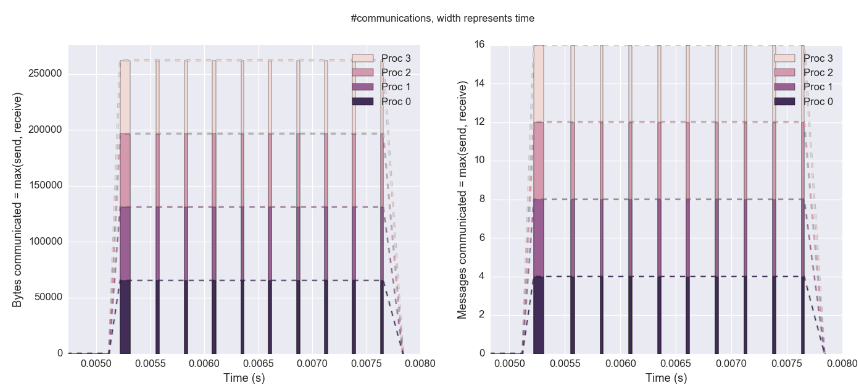


FIGURE 5.11: Profiler output for the FFT with $p = 4$ and $n = 8192$. Communication amount per superstep is shown on the vertical axis. The width of the bars and the gaps is an indication of the communication+synchronization time and the computation time respectively. The 10 bars represent 5 normal FFT and 5 inverse FFT computations, interchanged.

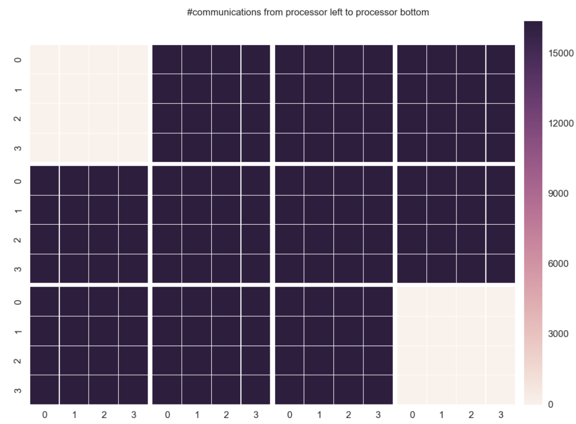


FIGURE 5.12: Profiler output for the FFT with $p = 4$ and $n = 8192$. Communication patterns for the processors are visualized. All FFT and inverse FFT computations require the same amount and same pattern of communication.

Conclusion

In most cases, the cost parameters of the library are not really taken into account until the last moment. In this thesis we have seen that a good implementation of communication and synchronization can even cut the execution time of an algorithm in half. Most speed is gained when the algorithm has only a small factor of difference in cost between computation and communication, like the FFT, in contrast with the LU decomposition that is computation bound by a factor of $O(n)$.

Some of the improvements done on Zefiros-BSPLib are more portable than others, when improving a distributed memory implementation. The Latin square is highly portable and is expected to give comparable improvements, while the Stack Allocators will most likely have less of an impact on distributed memory than it had on shared memory, because the information is already bundled and the actual exchange of information is far more expensive than the allocation of buffers.

The general idea is that abstraction in the code causes overhead in the runtime of the algorithm. For Zefiros-BSPLib, abstraction has proven useful in the optimization of the internals of the library.

The added Mixed Barrier in Zefiros-BSPLib is a nice addition for educational purposes. Correctness of the algorithm can be tested on more than the physically available cores that the computer actually has. It is also recommended to use the Mixed Barrier in the early stages of the algorithm development, as this is more processor-friendly than spinning indefinitely until everyone is done, in case of a computational imbalance. Using the profiler, we can then detect such imbalances and then try to fix them in our code.

Improvements can still be made on `Send`, as the time of this communication type is much larger than the other types of communication. This holds for both Zefiros-BSPLib and MulticoreBSP . Other improvements could be thought of that have not yet been mentioned in any form during this thesis. For example, synchronization in distinct subsets of the total processor set. This could be implemented in the form

```
BSPLib::Subdivide( /* subdivision definition */ ).
```

Extreme care needs to be taken in order to guarantee that the sets are really distinct, otherwise deadlocks that were not possible with the BSP standard could be reintroduced. The use of this is apparent from the LU decomposition matrix plot in Figure 5.7. In some supersteps, only pairs of processors need to communicate. It is redundant to wait for the other processors to finish before continuing on to the next superstep. If every processor follows the same pattern of subdivisions, this would never cause a deadlock. Feasibility of the subdivision should be checked for examples like the LU decomposition, to see if the communication pattern can easily enough be translated to a subdivision, before deciding if this is worthwhile.

Appendix A

C++ code for the parallel LU decomposition

Listing A.1: bsplu.cpp

```
#include "bspedupack.h"

#define EPS 1.0e-15

void bsp_broadcast( double *x, int n, int src, int s0, int stride, 5
                    int p0, int s, int phase )
{
  /* Broadcast the vector x of length n from processor src to
     processors s0+t*stride, 0 <= t < p0. Here n >= 0, p0 >= 1.
     The vector x must have been registered previously.
     Processors are numbered in one-dimensional fashion.
     s = local processor identity.
     phase= phase of two-phase broadcast (0 or 1)
     Only one phase is performed, without synchronization.
  */

  int b, t, t1, dest, nbytes;

  b = ( n % p0 == 0 ?  n / p0 : n / p0 + 1 ); /* block size */

  if ( phase == 0 && s == src )
  {
    for ( t = 0; t < p0; t++ )
    {
      dest = s0 + t * stride;
      nbytes = std::min( b, n - t * b ) * SZDBL;

      if ( nbytes > 0 )
      {
        bsp_put( dest, &x[t * b], x, t * b * SZDBL, nbytes );
      }
    }
  }

  if ( phase == 1 && s % stride == s0 % stride )
  {
    t = ( s - s0 ) / stride; /* s = s0+t*stride */

    if ( 0 <= t && t < p0 )
    {
      nbytes = std::min( b, n - t * b ) * SZDBL;

      if ( nbytes > 0 )
      {
        for ( t1 = 0; t1 < p0; t1++ )
        {
```

```
47            dest = s0 + t1 * stride;
48
49            if ( dest != src )
50            {
51              bsp_put( dest, &x[t * b], x, t * b * SZDBL, nbytes );
52            }
53          }
54        }
55      }
56    }
57
58  } /* end bsp_broadcast */
59
60  int nloc( int p, int s, int n )
61  {
62    /* Compute number of local components of processor s for vector
63       of length n distributed cyclically over p processors. */
64
65    return ( n + p - s - 1 ) / p ;
66
67  } /* end nloc */
68
69  void bsplu( int M, int N, int s, int t, int n, int *pi, double **a )
70  {
71    /* Compute LU decomposition of n by n matrix A with partial pivoting.
72       Processors are numbered in two-dimensional fashion.
73       Program text for P(s,t) = processor s+t*M,
74       with 0 <= s < M and 0 <= t < N.
75       A is distributed according to the M by N cyclic distribution.
76    */
77
78    int nloc( int p, int s, int n );
79    double *pa, *uk, *lk, *Max;
80    int nlr, nlc, k, i, j, r, *Imax;
81
82    nlr =  nloc( M, s, n ); /* number of local rows */
83    nlc =  nloc( N, t, n ); /* number of local columns */
84    bsp_push_reg( &r, SZINT );
85
86    if ( nlr > 0 )
87    {
88      pa = a[0];
89    }
90    else
91    {
92      pa = NULL;
93    }
94
95    bsp_push_reg( pa, nlr * nlc * SZDBL );
96    bsp_push_reg( pi, nlr * SZINT );
97    uk = vecallocd( nlc );
98    bsp_push_reg( uk, nlc * SZDBL );
99    lk = vecallocd( nlr );
100   bsp_push_reg( lk, nlr * SZDBL );
101   Max = vecallocd( M );
102   bsp_push_reg( Max, M * SZDBL );
103   Imax = vecalloci( M );
104   bsp_push_reg( Imax, M * SZINT );
105
106   /* Initialize permutation vector pi */
107   if ( t == 0 )
108   {
109     for ( i = 0; i < nlr; i++ )
110     {
```

```
111       pi[i] = i * M + s;   /* global row index */
112     }
113   }
114 
115   bsp_sync();
116   BSPProf::ResumeRecording();
117   BSPProf::MarkSuperstep( 0 );
118 
119   for ( k = 0; k < n; k++ )
120   {
121     int kr, kr1, kc, kc1, imax  = 0, smax, s1, t1;
122     double absmax, max, pivot;
123 
124     /* Initialise smax to non-existent index */
125     smax = -1;
126 
127     /****** Superstep 0 ******/
128     kr =  nloc( M, s, k ); /* first local row with global index >= k */
129     kr1 = nloc( M, s, k + 1 );
130     kc =  nloc( N, t, k );
131     kc1 = nloc( N, t, k + 1 );
132 
133     if ( k % N == t ) /* k=kc*N+t */
134     {
135       /* Search for local absolute maximum in column k of A */
136       absmax = 0.0;
137       imax = -1;
138 
139       for ( i = kr; i < nlr; i++ )
140       {
141         if ( fabs( a[i][kc] ) > absmax )
142         {
143           absmax = fabs( a[i][kc] );
144           imax = i;
145         }
146       }
147 
148       if ( absmax > 0.0 )
149       {
150         max = a[imax][kc];
151       }
152       else
153       {
154         max = 0.0;
155       }
156 
157       /* Broadcast value and local index of maximum to P(*,t) */
158       for ( s1 = 0; s1 < M; s1++ )
159       {
160         bsp_put( s1 + t * M, &max, Max, s * SZDBL, SZDBL );
161         bsp_put( s1 + t * M, &imax, Imax, s * SZINT, SZINT );
162       }
163     }
164 
165     bsp_sync();
166     BSPProf::MarkSuperstep();
167 
168     /****** Superstep 1 ******/
169     if ( k % N == t )
170     {
171       /* Determine global absolute maximum (redundantly) */
172       absmax = 0.0;
173 
174       for ( s1 = 0; s1 < M; s1++ )
```

```
175        {
176          if ( fabs( Max[s1] ) > absmax )
177          {
178            absmax = fabs( Max[s1] );
179            smax = s1;
180          }
181        }
182
183        if ( absmax > EPS )
184        {
185          r = Imax[smax] * M + smax; /* global index of pivot row */
186          pivot = Max[smax];
187
188          for ( i = kr; i < nlr; i++ )
189          {
190            a[i][kc] /= pivot;
191          }
192
193          if ( s == smax )
194          {
195            a[imax][kc] = pivot;  /* restore value of pivot */
196          }
197
198          /* Broadcast index of pivot row to P(*,*) */
199          for ( t1 = 0; t1 < N; t1++ )
200          {
201            bsp_put( s + t1 * M, &r, &r, 0, SZINT );
202          }
203        }
204        else
205        {
206          bsp_abort( "bsplu at stage %d: matrix is singular\n", k );
207        }
208      }
209
210      bsp_sync();
211      BSPProf::MarkSuperstep();
212
213      /****** Superstep 2 ******/
214      if ( k % M == s )
215      {
216        /* Store pi(k) in pi(r) on P(r%M,0) */
217        if ( t == 0 )
218        {
219          bsp_put( r % M, &pi[k / M], pi, ( r / M )*SZINT, SZINT );
220        }
221
222        /* Store row k of A in row r on P(r%M,t) */
223        bsp_put( r % M + t * M, a[k / M], pa, ( r / M )*nlc * SZDBL, nlc * SZDBL );
224      }
225
226      if ( r % M == s )
227      {
228        if ( t == 0 )
229        {
230          bsp_put( k % M, &pi[r / M], pi, ( k / M )*SZINT, SZINT );
231        }
232
233        bsp_put( k % M + t * M, a[r / M], pa, ( k / M )*nlc * SZDBL, nlc * SZDBL );
234      }
235
236      bsp_sync();
237      BSPProf::MarkSuperstep();
238
```

```
239     /****** Superstep 3 ******/
240     /* Phase 0 of two-phase broadcasts */
241     if ( k % N == t )
242     {
243       /* Store new column k in lk */
244       for ( i = kr1; i < nlr; i++ )
245       {
246         lk[i - kr1] = a[i][kc];
247       }
248     }
249 
250     if ( k % M == s )
251     {
252       /* Store new row k in uk */
253       for ( j = kc1; j < nlc; j++ )
254       {
255         uk[j - kc1] = a[kr][j];
256       }
257     }
258 
259     bsp_broadcast( lk, nlr - kr1, s + ( k % N )*M,  s, M, N, s + t * M, 0 );
260     bsp_broadcast( uk, nlc - kc1, ( k % M ) + t * M, t * M, 1, M, s + t * M, 0 );
261     bsp_sync();
262 
263     BSPProf::MarkSuperstep();
264 
265     /****** Superstep 4 ******/
266     /* Phase 1 of two-phase broadcasts */
267     bsp_broadcast( lk, nlr - kr1, s + ( k % N )*M,  s, M, N, s + t * M, 1 );
268     bsp_broadcast( uk, nlc - kc1, ( k % M ) + t * M, t * M, 1, M, s + t * M, 1 );
269     bsp_sync();
270 
271     BSPProf::MarkSuperstep( 0 );
272 
273     /****** Superstep 0 ******/
274     /* Update of A */
275     for ( i = kr1; i < nlr; i++ )
276     {
277       for ( j = kc1; j < nlc; j++ )
278       {
279         a[i][j] -= lk[i - kr1] * uk[j - kc1];
280       }
281     }
282   }
283 
284   BSPProf::PauseRecording();
285 
286   bsp_pop_reg( Imax );
287   vecfreei( Imax );
288   bsp_pop_reg( Max );
289   vecfreed( Max );
290   bsp_pop_reg( lk );
291   vecfreed( lk );
292   bsp_pop_reg( uk );
293   vecfreed( uk );
294   bsp_pop_reg( pi );
295   bsp_pop_reg( pa );
296   bsp_pop_reg( &r );
297 
298 } /* end bsplu */
```

LISTING A.2: bsplu_test.cpp

```
#include "bspedupack.h"

/*  This is a test program which uses bsplu to decompose an n by n
  matrix A into triangular factors L and U, with partial row pivoting.
  The decomposition is A(pi(i),j)=(LU)(i,j), for 0 <= i,j < n,
  where pi is a permutation.

  The input matrix A is a row-rotated version of a matrix B:
    the matrix B is defined by: B(i,j)= 0.5*i+1    if i<=j
                    0.5*j+0.5    i>j,
    the matrix A is defined by: A(i,j)= B((i-1) mod n, j).

  This should give as output:
    the matrix L given by: L(i,j)= 0   if i<j,
                   = 1    i=j,
                   = 0.5  i>j.
    the matrix U given by: U(i,j)= 1   if i<=j,
                   = 0    i>j.
    the permutation pi given by:  pi(i)= (i+1) mod n.

  Output of L and U is in triples (i,j,L\U(i,j)):
    (i,j,0.5) for i>j
    (i,j,1)   for i<=j
  Output of pi is in pairs (i,pi(i))
    (i,(i+1) mod n) for all i.

  The matrix A is constructed such that the pivot choice is unique.
  In stage k of the LU decomposition, row k is swapped with row r=k+1.
  For the M by N cyclic distribution this forces a row swap
  between processor rows.
*/

uint32_t M, N;

void bsplu_test()
{
  int nloc( int p, int s, int n );
  void bsplu( int M, int N, int s, int t, int n, int *pi, double **a );
  int p, pid, q, s, t, n, nlr, nlc, i, j, iglob, jglob, *pi;
  double **a, time0, time1;

  bsp_begin( M * N );
  BSPProf::PauseRecording();
  p = bsp_nprocs(); /* p=M*N */
  pid = bsp_pid();

  bsp_push_reg( &M, SZINT );
  bsp_push_reg( &N, SZINT );
  bsp_push_reg( &n, SZINT );
  bsp_sync();

  if ( pid == 0 )
  {
    printf( "Please enter matrix size n:\n" );

    //*
#ifdef _WIN32
    scanf_s( "%d", &n );
#else
    scanf( "%d", &n );
#endif
    /*/
```

```
63      n = 100;
64      /**/
65
66      for ( q = 0; q < p; q++ )
67      {
68        bsp_put( q, &M, &M, 0, SZINT );
69        bsp_put( q, &N, &N, 0, SZINT );
70        bsp_put( q, &n, &n, 0, SZINT );
71      }
72    }
73
74    bsp_sync();
75    bsp_pop_reg( &n ); /* not needed anymore */
76    bsp_pop_reg( &N );
77    bsp_pop_reg( &M );
78
79    /* Compute 2D processor numbering from 1D numbering */
80    s = pid % M; /* 0 <= s < M */
81    t = pid / M; /* 0 <= t < N */
82
83    /* Allocate and initialize matrix */
84    nlr =  nloc( M, s, n ); /* number of local rows */
85    nlc =  nloc( N, t, n ); /* number of local columns */
86    a = matallocd( nlr, nlc );
87    pi = vecalloci( nlr );
88
89    if ( s == 0 && t == 0 )
90    {
91      printf( "LU decomposition of %d by %d matrix\n", n, n );
92      printf( "using the %d by %d cyclic distribution\n", M, N );
93    }
94
95    for ( i = 0; i < nlr; i++ )
96    {
97      iglob = i * M + s;  /* Global row index in A */
98      iglob = ( iglob - 1 + n ) % n; /* Global row index in B */
99
100     for ( j = 0; j < nlc; j++ )
101     {
102       jglob = j * N + t; /* Global column index in A and B */
103       a[i][j] = ( iglob <= jglob ? 0.5 * iglob + 1 : 0.5 * ( jglob + 1 ) );
104     }
105   }
106
107   if ( s == 0 && t == 0 )
108   {
109     printf( "Start of LU decomposition\n" );
110   }
111
112   bsp_sync();
113   time0 = bsp_time();
114
115   bsplu( M, N, s, t, n, pi, a );
116   bsp_sync();
117   time1 = bsp_time();
118
119   if ( s == 0 && t == 0 )
120   {
121     printf( "End of LU decomposition\n" );
122     printf( "This took only %.6lf seconds.\n", time1 - time0 );
123     printf( "\nThe output permutation is:\n" );
124     fflush( stdout );
125   }
126
```

```
127   if ( t == 0 )
128   {
129     for ( i = 0; i < nlr; i++ )
130     {
131       iglob = i * M + s;
132       printf( "i=%d, pi=%d, proc=(%d,%d)\n", iglob, pi[i], s, t );
133     }
134 
135     fflush( stdout );
136   }
137 
138   bsp_sync();
139 
140   if ( s == 0 && t == 0 )
141   {
142     printf( "\nThe output matrix is:\n" );
143     fflush( stdout );
144   }
145 
146   for ( i = 0; i < nlr; i++ )
147   {
148     iglob = i * M + s;
149 
150     for ( j = 0; j < nlc; j++ )
151     {
152       jglob = j * N + t;
153       printf( "i=%d, j=%d, a=%f, proc=(%d,%d)\n",
154           iglob, jglob, a[i][j], s, t );
155     }
156   }
157 
158   vecfreei( pi );
159   matfreed( a );
160 
161   bsp_end();
162 }
163 
164 int main( int argc, char **argv )
165 {
166 
167   bsp_init( bsplu_test, argc, argv );
168 
169   printf( "Please enter number of processor rows M:\n" );
170 
171   //*
172 #ifdef _WIN32
173   scanf_s( "%d", &M );
174 #else
175   scanf( "%d", &M );
176 #endif
177 
178   printf( "Please enter number of processor columns N:\n" );
179 
180 #ifdef _WIN32
181   scanf_s( "%d", &N );
182 #else
183   scanf( "%d", &N );
184 #endif
185   /*/
186   M = 8;
187   N = 1;
188   /**/
189 
190   if ( M * N > bsp_nprocs() )
```

```
191   {
192     printf( "Sorry,␣not␣enough␣processors␣available.\n" );
193     fflush( stdout );
194     exit( 1 );
195   }
196
197   bsplu_test();
198   exit( 0 );
199
200 } /* end main */
```

Bibliography

[1] R. H. Bisseling, *Parallel scientific computation. a structured approach using bsp and mpi*. Oxford University Press, Mar. 2004, ISBN: 9780198529392.

[2] A. N. Yzelman, R. H. Bisseling, D. Roose, and K. Meerbergen, “MulticoreBSP for C: A high-performance library for shared-memory parallel programming”, *International journal of parallel programming*, vol. 42, no. 4, pp. 619–642, 2014.

[3] J. M. D. Hill and D. B. Skillicorn, “Lessons learned from implementing BSP”, *Future generation computer systems*, vol. 13, pp. 327–335, 1997/1998.

[4] J. M. Hill, *Oxford bsp toolset*, http://www.bsp-worldwide.org/implmnts/oxtool/, 1998.

[5] J. W. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex Fourier series”, *Mathematics of computation*, vol. 19, pp. 297–301, 1965.