Extending the BSP model to hierarchical heterogeneous architectures

Thesis

M. van Duijn

Scientific Computing Group
Mathematical Institute
Utrecht University

Supervised by
Prof. dr. R.H. Bisseling

Universiteit Utrecht

June 23, 2018

Abstract

In the current field of High Performance Computing (HPC), the growing scale and complexity of problems creates a growing need for scalable parallel algorithms. To achieve scalability, we ideally want to make use of as many processors, as efficiently as possible. Many models have been (and are still) developed to describe system architectures, and to extend these models for more optimal use of newer architectures. These models form the framework for algorithm development and analysis. In this research project, an effort was made to generalise the Bulk Synchronous Parallel (BSP) model, a mathematical model for constructing and analysing parallel algorithms, to hierarchical heterogeneous architectures.

Acknowledgements

I would like to thank my supervisor Rob Bisseling for his guidance and inspiration during this thesis. His course on Parallel Algorithms at the start of the master Mathematical Sciences sparked my interest in the BSP model. He has also provided inspiration for the High-Performance LU decomposition developed in this thesis.

Contents

1 Introduction

In this thesis, an overview will be given of the many different system architectural properties of (highly) parallel machines and the models that are used to analyse them. These models are vital tools to analyse high performance algorithms, in terms of speed and scalability. High Performance Computing is applied in many scientific fields such as (Bio)chemistry, Environmental Modelling regarding weather, climate and earthquakes, Artificial Intelligence, Finance, and many more.

The BSP model, a model that has proven its value, will be discussed more in-depth, and the generalisations that are made by the model that impact analysis and performance for hierarchical heterogeneous machines will be identified. In this thesis, an effort is made to extend the model to better account for these properties, and an accompanying software library SyncLib is presented that implements these extensions for utilisation in the algorithms. This extension will be both for the mathematical analysis of algorithms, as well as the implementation of algorithms, and this will be the main goal for this thesis. Part of this goal is to measure the heterogeneous properties of the machine, and to try find the best fit to existing communication patterns, to further decrease communication cost. The extensions that are proposed and implemented in this thesis, are at the basis a combination of improvements that have either been theoretically or experimentally studied in literature. By studying multiple versions of all the improvements, the lessons learned in other research are bundled and refined to create an extension to the existing BSP model.

Finally two linear algebra algorithms - matrix multiplication and a high-performance variant of LU decomposition with partial pivoting - will be discussed both analytically, and experimentally on a supercomputer. Linear algebra algorithms are vital building blocks for many larger algorithms, so it is important that these are well-optimised and scalability is well understood.

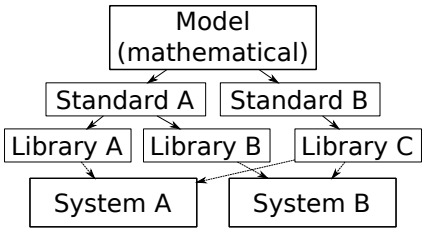
In Chapter 2, a brief introduction to the properties and restrictions of parallel computing is given. Important concepts that impact what can an cannot be assumed in parallel computing models are explained. Existing models and libraries for different architectures are discussed. In Chapter 3, we will discuss an entry-level, yet important parallel algo-

rithm: parallel matrix-matrix multiplication. The algorithm is laid out and the time complexity is analysed using the BSP model. Chapter 4 addresses some of the generalisations made in the BSP model that could be further specialised to fit better to modern supercomputer architectures. We introduce some terminology for mathematical analysis for the extensions proposed. In Chapter 5 we discuss a benchmark framework for non-hierarchical machines, and use some of its components to devise a benchmark for hierarchical and heterogeneous properties of communication in the machine. Then we propose an algorithm to fit these properties to the communication patterns present in many parallel algorithms. Chapter 6 introduces a novel `C++` library, SyncLib, to incorporate the proposed improvements into algorithm development, in a portable manner. This library puts a modern twist on the programming standard that accompanies the BSP model, to better fit an Object-Oriented programming style in `C++`, and allows for the hierarchy of the machine to be exploited if present. In Chapter 7 we will use the extended analysis framework to analyse the aforementioned two linear algebra algorithms - matrix-matrix multiplication and LU decomposition - to demonstrate how the possible hierarchies can be dynamically incorporated into the time complexity analysis. In Chapter 8 we will perform experiments with both algorithms on Cartesius, the Dutch national supercomputer. Several configurations to address the hierarchy are put to the test, and results are compared to the peak performance reached per core in the machine. Finally, we will summarise what was discussed and draw conclusions on what extensions of the model were most beneficial to performance. We will discuss and suggest improvements that can still be made.

2 Parallel Computing

This chapter provides an overview of parallel computing in several levels of abstraction. In this context we will view a processor as a single computational unit that executes its own part of the computation. When developing a parallel algorithm, we want to distribute the work, and therefore require data sharing and synchronisation. Distributing the work means that every intermediate result computed on the processors is not immediately accessible to other processors. In almost any parallel algorithm, there is a point at which the intermediate results need to be combined. This is when we make use of synchronisation and data sharing.

Beside scalability, we also desire portability for algorithms. We do not want to redesign the entire algorithm if a new type of hardware pops up with slightly different properties. Parallel computing can be viewed in several layers of abstraction.



A model can be used to mathematically analyse a problem. A model builds on assumptions, that can then be provided by a standard. A model is used to analyse the expected computation time and communication volume, and to determine the expected order of speedup when using more processors in parallel. In turn, a standard defines the guarantees the library has to provide, and the way the library has to expose these functionalities. Libraries in turn take the properties of one or multiple system architectures into consideration, and properly expose or simulate the guarantees to adhere to the standard. The system architecture gives limitations to which guarantees are readily available, and which have to be solved in software by the library developer. By constructing an algorithm based on a model, the only thing that is needed on a new architecture is a library adhering to the standard and then we can run all of our existing algorithms on it.

2.1 General system architectural properties

To better understand the guarantees we can and want to give in a model, we first have to analyse the system architectural properties that are available. Ideally, we want a model that is independent of system architecture. Due to the large differences, this either requires a generalisation to common properties, or simulation of missing properties. Generalisation of properties often leads to suboptimal use of the system architecture, and simulation leads to overhead.

2.1.1 Synchronisation

There are several flavours of synchronisation. Processors can operate synchronously or asynchronously, and synchronisation can be provided in an asynchronous environment to facilitate some deterministic order in the program. If we look at the development in processors, the latter is far more widely adopted, so we will focus on synchronisation of asynchronous environments. This type of synchronisation can happen pairwise, on a subset of the processors or globally. Synchronisation of a subset of the processors means that the state of this subset of processors is guaranteed just after the synchronisation. Synchronisation is useful for proving and guaranteeing correctness of an algorithm that is largely executed asynchronously. Synchronisation in asynchronous computers is often a grey area between system architecture and software, and is therefore solved by the library developer.

2.1.2 Data sharing

There are two main categories of data sharing: data sharing in a shared memory environment, and data sharing in a distributed memory environment. Local memory refers to the memory only accessible to a single processor in both shared- and distributed-memory environments.

Shared memory

In a shared memory environment, there is some global memory accessible to all processors. To share data, one processor writes to the shared memory, and after synchronisation, the other processor reads from this shared memory. One caveat of shared memory is proving consistency of memory access patterns. If one processor starts to write a block in the shared memory, and another tries to read or write part of that block at the same time, correctness of an algorithm is hard to guarantee. Ways to overcome this problem are locking, dedicating memory to one processor before synchronisation and atomic operations.

Locking and dedicating memory are very similar, because one processor can access a specific part of the memory at a time. Often dedicating memory is done implicitly by the user of a standard, by writing the program in such a way that only one processor uses any part of the

memory at the same time, before synchronisation. Locking on the other hand is very explicit. A processor locks the memory and has exclusive access to it for the duration of the lock. Locking is useful when operations on this shared memory are sparse throughout the algorithm. However, it gives rise to the dreaded deadlock. A deadlock occurs when locks are nested in a different order over several processors. Processors have a resource locked, and are waiting for other processors to release another resource. This can be pairwise or in some cyclic fashion with more processors. The third solution, atomic operations, guarantees that one read ore write action is always completed before the next is started. Composite operations like fetch-and-increase or fetch-and-decrease are sometimes also provided. This is useful for when the operations are commutative. Due to the complex nature of atomic operations, they are often much more time-consuming than normal computations, so they should be used sparsely. Without an atomic operation, the usual way to modify a variable is to temporarily load it into local memory, then apply several operations and write back the result. If multiple processors do this concurrently, they read the initial result into their local memory, without modifications by the other processors, and perform their local operations. When they all write back the result, the resulting value is only affected by the processor that is last in writing back the result, which is undesired behaviour, and we need to pay attention to this in shared memory environments. There are models to guide these access patterns, but not always safeguards that prevent incorrect use.

Another caveat of shared memory is that, due to hardware restrictions, providing shared memory to a lot of processors introduces a lot of overhead, so there is usually a relatively small number of processors with access to the shared memory.

Distributed memory

In a distributed memory environment, each processor only has access to its own local memory. Sharing data has to be done through a communication network. This network is often viewed as a black box, and standards and models only define the behaviour and guarantees it gives us about how and when the communication is completed. Distributed memory also means you need to take into account that not everyone has access

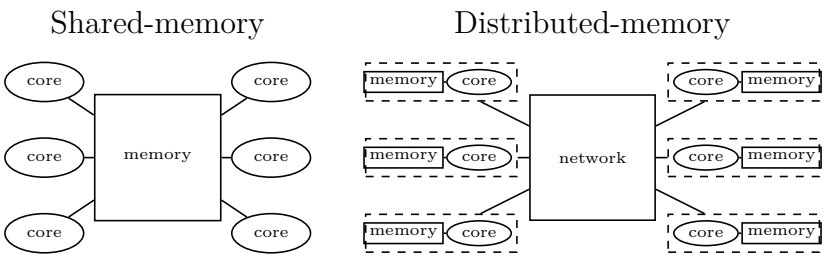


Figure 2.1: A visual representation of shared- vs. distributed-memory.

to all the input data. Different data distributions can greatly influence performance.

Communication can be separated into two categories: one-sided communication and two-sided communication. This refers to the number of active participants in the communication. In one-sided communication, either the sending or receiving party actively participates in the communication, while the other passively participates. No confirmation of sending or receiving is needed. In two-sided communication, the sending and receiving party have to both confirm the action of the other. If done improperly, this can again result in the dreaded deadlock situation, similar to locking. One-sided communication is therefore much easier to use, since only one party needs to know the desired communication, and the risk of deadlocks is eliminated.

2.2 Hybrid system architectures

In reality, most systems have properties that are a hybrid of the aforementioned properties. Even on a small scale, your desktop computer or laptop is a hybrid of shared and distributed memory. In terms of these properties, it consists of a Central Processing Unit (CPU) and some Random Access Memory (RAM). The CPU consists of multiple cores (processors), each equipped with a small cache. This cache is distributed, while the RAM is shared. However, the difference here is more subtle, as your data is automatically written back and forth between cache and RAM, without direct control. On a larger scale, a supercomputer usually consists of nodes, each equipped with a CPU. There are two levels of distributed memory: RAM is distributed over nodes, and cache is distributed over processors of the CPU, but RAM is shared between processors of the same CPU. Data sharing between nodes can only be done through a communication network. A node of a supercomputer can also contain multiple CPUs, which introduces yet another level of distributed memory: a cache shared by processors on the same CPU, but not between the multiple CPUs. Then there are still other types of processor chips, generally

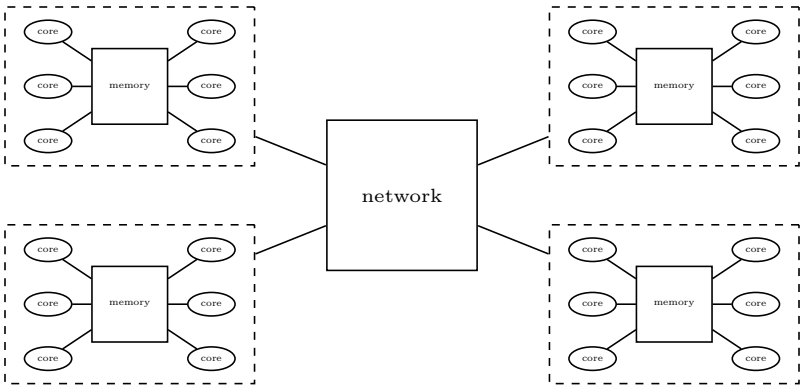


Figure 2.2: A visual representation of hybrid distributed shared-memory.

called co-processors. Popular co-processors are the Graphics Processing Unit (GPU) and General Purpose Graphics Processing Unit (GPGPU), and the Intel® Xeon Phi co-processor line. GPUs are originally developed for graphics computations, but as those computations are usually matrix-matrix and other linear algebra operations, they can be efficiently used in applications unrelated to graphics that utilise those types of computations. Rising interest in fields such as Artificial Intelligence (AI) and cryptocurrency further increase the popularity of the GPU for High Performance Computing (HPC). More exotic processor chips are the Application Specific family of chips, designed for a specific computation, and many-core co-processors like the Epiphany co-processor, aimed at bringing supercomputing to a price level affordable for hobbyists.

This only covers a small subset of systems, but as we can see, the list of different architectures is already very large, and only growing with the technological developments. No model can account for all future changes at once, without making generalisations of the architecture.

2.3 Existing Models

Models describe a generalisation of some architecture, a computer, and parametrise properties of the computer to analyse the expected speedup behaviour of using more processors in the computer. As the architecture of the memory determines the efficiency and means of communication, there are two main categories of models: those based on shared memory, and those based on distributed memory. They have grown alongside the physical developments in parallel computing, both to describe and to guide parallel architectures. Then there are also hybrid models. Those are usually extensions of some distributed memory model, and they account for the physical existence of memory and network hierarchies.

2.3.1 PRAM

The Parallel Random Access Machine (PRAM), as the name suggests, is a model that generalises some parallel shared memory machines [Wyl79]; [JáJ92]; [KKT01]. There are different flavours of the PRAM model, accounting for the different strategies to handle concurrent read and write operations. The PRAM model makes several simplifying assumptions. A PRAM consists of

1. An unbounded set of processors $P_0, P_1, \ldots$, each with an unbounded local memory.

2. An unbounded global memory, uniformly accessible by any processor.

3. A set of input registers

4. A finite program, consisting of synchronous Single Instruction Multiple Data (SIMD) operations.

While the model provides the basis for analysis of parallel algorithms, the last assumption is often not realistic on a larger scale, both in the hardware implementation and in algorithm development. The model does not account for the increased access time of the shared memory, so while it can be used to analyse the speedup behaviour, it is often not a realistic representation of the underlying hardware. Some extensions have been made to account for this. The Local-memory Parallel Random Access Machine (LPRAM), first proposed in 1990 by Aggarwal, Chandra, and Snir [ACS90], is such an extension. It takes into account the difference between local and global memory, and captures this in terms of communication volume between global and local memory. This gives a more realistic bound on the speedup.

2.3.2 BSP Computer

The Bulk Synchronous Parallel (BSP) computer is a bridging model for writing parallel algorithms, aimed at generalising many parallel computers, only assuming the existence of asynchronous processors, and a communication network. This communication network provides means of bulk communication and bulk synchronisation. The bulk synchronisation is achieved through explicit barrier synchronisation: every processor has to reach the barrier, before they all continue. The BSP model was first introduced by Valiant in 1990 [Val90], and was aimed to overcome the gap between theoretic cost and experimental cost, and to guide hardware designers by providing a model similar to that for sequential computers. The BSP model has undergone some iterations of changes to facilitate easier analysis of algorithms and more realistic cost predictions. The version that will be used in this thesis is the version proposed by Bisseling and McColl [BM94] in 1993, which has been the basis for development of a programming standard, BSPlib [Hil+98] in 1998. The proposed version parametrises a BSP computer by the following parameters.

1. r: the single-processor computation rate. An expected number of floating point operations per second.

2. g: the communication throughput ratio. An expected number of floating point operations performed to communicate one data word.

3. l: the synchronisation cost. The minimum overhead for bulk synchronisation before every processor can continue. This consists of the constant overhead for synchronisation, like network latency.

4. p: the number of available processors in the BSP computer.

All processors in the BSP computer are assumed to have uniform performance, and communication cost and synchronisation cost is also

assumed to be uniform. If this assumption does not hold, there will be some penalty in the performance as compared to the optimal performance. However, this is often deemed acceptable as cost for the ease of analysis.

In the original model, a different parameter was used instead of the network latency, namely superstep periodicity L: in each superstep, a maximum of L local operations can be performed, or L/g data words can be communicated. This however imposes a structure on an algorithm that is very different from the usual sequential algorithm.

The BSP variant proposed by Bisseling and McColl for parallel programming divides an algorithm into supersteps that are either purely computational, with only local operations, or purely communicational. The cost of a computation superstep with w floating point operations is $w + l$. In a communication superstep, we analyse $h_{\text{send}}, h_{\text{receive}}$ as the maximum number of sent and received data words by a single processor. As sending and receiving is done in bulk, and one is always passive, the cost of such a superstep can be expressed as $hg + l$, where $h = \max(h_{\text{send}}, h_{\text{receive}})$. The total cost of an algorithm consisting of m supersteps then simply becomes

$$\sum_{j=1}^{m}(w_j + h_j \cdot g + l) = W + H \cdot g + m \cdot l,$$

where $W = \sum_{j=1}^{m} w_j$ and $H = \sum_{j=1}^{m} h_j$, where we simply put $w_j = 0$ in case it is a communication superstep, and $h_j = 0$ in case it is a computation superstep. With this cost expression, and measured values for (r, g, l), we can analytically determine what is expected to be the optimal machine configuration. How these parameters are measured, is later explained in Chapter 5.

2.3.3 LogP Machines

The Latency overhead gap Processors (LogP) machine, proposed in 1993 [Cul+93]; [Cul+96], is very similar to the BSP machine, as it describes a similar architecture. Parameters are similar to the model proposed by Bisseling and McColl. It addresses the problem stated before: the original BSP model, while being an improvement over the PRAM, is radically different than the usual way of modelling (sequential) algorithms, and imposes an algorithm structure that is not always trivial to construct. Instead of expressing superstep periodicity, the model describes a parallel computer with the following parameters:

1. L: the network latency. Constant overhead for sending a word, or small number of words.

2. o: the communication overhead. The time it takes to transmit a single data word.

3. g: the gap between messages before the processors are ready to send or receive again.

4. P: the number of available processors.

As we can see, these parameters are very similar to the parameters proposed by Bisseling and McColl. Of course, we can incorporate the r-parameter, the single-processor computation rate, to normalise the number of operations to expected execution time. Furthermore, there is an assumed limit to the network capacity, such that $\lceil L/g \rceil$ can be in transit in total at any time. Exceeding this capacity means the gap g is incurred before further communication continues.

The main difference of this model with respect to the BSP model, is the departure from the bulk communication and synchronisation idiom. While providing more fine-grained control over communication patterns, allowing communication between processors in a subset to overlap with computation of a different (disjoint) subset of processors, it complicates the analysis of an algorithm as a whole, and can lead to unexpected behaviour in case some processors slow down. An analysis and comparison of (asymptotic) performance is made in [Bil+96]. This analysis concludes that the asymptotic behaviour is very similar, and that under certain circumstances, the two models can simulate each other with constant overhead. It is also argued that the BSP model provides a far more convenient abstraction in terms of algorithm analysis, because the algorithm designer does not have to take into account the network capacity. It is also argued that in the LogP framework, due to the lack of bulk synchronisation, a change of machine parameters can turn a correct algorithm into an incorrect algorithm, because the execution order may be highly reliant on those parameters, whereas the BSP model has a deterministic order due to supersteps.

2.3.4 BSP extensions

Over the years, there have been many extension, variants and adaptations of the BSP model. They mostly account for memory hierarchies, network hierarchies, and some account for exotic architectures.

BSPRAM

The Bulk Synchronous Parallel Random Access Machine (BSPRAM) model is, as the name suggests, a merger of the BSP model and the PRAM model [Tis98]. It is a variant of the regular BSP model. Instead of a communication network, the model considers some global shared memory. Instead of communication volume, it considers the volume of reads and writes to the global memory. Like the PRAM model, it has several flavours of what type of concurrent memory access is allowed. Algorithm design is the same as the BSP paradigm, in supersteps. The aim is to simplify communication patterns, by making a processor oblivious to the owner of the data. With some small overhead, this can be efficiently simulated on a regular BSP computer without the explicit existence of a global shared memory, by communicating changes in ownership, and

letting the model determine communication patterns from this knowledge about ownership. What is also gained by this abstraction, is that the availability of this global shared memory in the physical architecture may further reduce the communication cost. The discussed LPRAM model may be viewed as an early attempt at this model, but without the bulk-synchronisation capabilities.

D-BSP and E-BSP

The Decomposable BSP (D-BSP) and Extended BSP (E-BSP) models are an extension of the BSP model, that were proposed to take into account hierarchies in the network. The D-BSP was first introduced in [DK96]. It proposes to divide the available processors into clusters, where we call a cluster a submachine. This clustering is done in such a way, that the submachine has a lower communication and synchronisation cost than the overall worst-case costs. It can be viewed as a more formal definition of the subset synchronisation proposed in the original model by Valiant[Val90]. The E-BSP model views the network architecture as a whole, and provides a more fine grained communication cost prediction given the participating processors. The cost can be viewed in terms of the shortest distance between two processors in the network. A later survey in [Bil+01] on the effectiveness of these models as a bridging model discusses both models, but favours the D-BSP model for being more general and providing more abstract analysis. It argues the clustering is sufficient to capture proximity in the network. It shows some promising algorithms and network topologies that greatly benefit from the clustering based on the D-BSP model.

Multi-BSP

One of the extensions accounting for memory hierarchies, proposed by Valiant himself [Val11], is the Multi-BSP model. Instead of modelling a BSP computer as individual processors, the computer is modelled hierarchically, and each level j in the hierarchy has 4 parameters (p_j, g_j, L_j, m_j), where the first three are the same parameters as in the original model, but m_j accounts for the available memory at that level. It can be viewed as a more general, better scalable version of the BSPRAM model, with an arbitrary number of levels of shared memory, each introducing slower access times. A BSPRAM machine with parameters (p, g, L), shared memory of size m_2, and local memory of size m corresponds to a Multi-BSP machine with parameters $(p_1 = 1, g_1 = g, L_1 = 0, m_1 = m)(p_2 = p, g_2 = \infty, L_2 = L, m_2)$ [Val11]. The leaves (level 1) consist of a single processor $p_1 = 1$, and have local memory of size $m_1 = m$. Synchronisation in the leaves is only with one processor, so $L_1 = 0$. Writing to the shared memory on the higher level has cost $g_1 = g$. In the root (level 2), synchronisation of the child nodes has cost $L_2 = L$. There are $p_2 = p$ child nodes. Writing to memory on a higher level is not possible, so $g_2 = \infty$ and finally, the shared

memory size is m_2. The Multi-BSP model can be viewed as a restricted, recursive application of the D-BSP model, where each level is a uniform clustering of processors. While this model can theoretically describe an entire data-center, the arbitrary depth of the hierarchy complicates construction and analysis of algorithms.

2.4 Libraries

Many libraries have been developed to support parallel computing. Most notable for portability are the MPI standard library [MPI16] and OpenMP [Ope17]. Most programming languages also provide (explicit) Multi-Threading (MT).

2.4.1 MPI

The Message Passing Interface (MPI) standard originally defined communication as two sided, using send and receive primitives. Algorithms written directly using the MPI are perhaps best analysed using the LogP model, because of the fine-grained control over communication. As discussed, analysis in this model is not always easily constructed, and the two-sided communication easily leads to deadlocks if used incorrectly.

2.4.2 OpenMP and Multi-Threading

OpenMP and MT both work on single machines with multiple processors (cores). As such, they rely on shared memory for data sharing. Direct implementations in OpenMP and MT probably fit best to either the LPRAM or BSPRAM model, but they lack easy synchronisation primitives in their "vanilla" form. The LPRAM analysis only works if processors are regularly synchronised after writing to or before reading from the shared memory. When using MT, it is important to look at portability. Often system specific MT is faster, but not portable. It can be used, but with a fallback that is supported on a wider range of systems, like the C++ standard library for threading, Intel® Threading Building Blocks (Intel® TBB)[Int17] and OpenMP.

2.4.3 BSP libraries

The BSP model has been realised in many libraries. Many early libraries are direct implementations of the BSPlib[Hil+98] standard, including the Oxford BSP library [HDM98]. It is also possible to add an abstraction layer to MPI or Multi-Threading libraries to support the bulk parallel paradigm, as is shown in the BSPonMPI library[Sui06]. Due to the popularity and wide availability of MPI, it is a very attractive portability layer, because it instantly supports compilation on a very wide range of architectures. Efficient shared memory implementations include MulticoreBSP **multicorebsp** and Zefiros BSPLib [DVV17]. These could be

used in combination with an MPI based library to implement a Multi-BSP algorithm. The more recent version of MulticoreBSP supports both modes of operation, and supports nested runs to facilitate Multi-BSP algorithms. A more modern approach that is a slight departure from the standard, is the Bulk library [BB17]; [BBB]. It is intended to provide higher level memory management, similar to what the C++ standard provides over the C standard, by providing predefined shared container types, and a more Object Oriented (OO) approach to registration of shared variables, such as Resource Acquisition Is Initialisation (RAII), which relieves the developer from memory cleanup. Bulk also supports different communication backends, and nested environments similar to MulticoreBSP, and allows for custom backend development for future communication technologies.

Another attempt at hybridisation of MPI and OpenMP architectures is the BSP++ library [HFE10]. Instead of implementing the BSPlib standard, it defines its own notion of parallel data structures, that can be split or accumulated. It does show very interesting experimental results, showing good speedups with little effort. Future aims included GPU versions of the library, but they have not yet been published.

The NestStep [Keß00] programming language can be viewed as a realisation of subset synchronisation in the BSP model. The language makes it possible to dynamically split the processors into disjoint subsets, and to perform a nested BSP program on each subset, with synchronisation being global to the subset, but not all subsets. Global synchronisation is then achieved by rejoining the subsets. With proper management of subsets, it could potentially be used to implement a D-BSP algorithm, where each submachine (cluster) can be represented by a processor subset. Proper management of hierarchies is needed for successful D-BSP implementations.

BSP libraries for more exotic architectures have also popped up, such as the Epiphany BSP [BBW15]; [BBW16] library, developed for BSP-like algorithms on the Parallella [Par14]. The difference with regular BSP algorithms is that it needs to be written in a streaming fashion for larger input sizes. This means not every BSP program can be directly executed using Epiphany BSP. Some modifications need to be made to support streaming from the main processor to the co-processor. Another such library is the Bulk Synchronous GPU Programming (BSGP) library. It was intended to be a higher level language alternative to Compute Unified Device Architecture (CUDA) code [NVI07], the language offered by NVIDIA. It supports BSP-style programming on the GPU, using a custom pre-compiler that compiles the C-like BSGP language to C and CUDA code.

As the title suggests, the BSP model will again be the basis for the extended model, taking into account some of the lessons learned from other efforts, and combining ideas from the different models.

3 Parallel matrix-matrix multiplication

To gain some familiarity with the BSP model, we will analyse one of the building blocks of many HPC algorithms: the matrix-matrix multiplication. There is a wide variety of choices when it comes to (parallel) matrix multiplication. Some focus on precision, others on optimising the computational complexity. In parallel matrix multiplication, there is also a trade-off between using replication through using extra memory, and the communication time. Some algorithms work for variable size matrices, others only for square matrices, or even square matrices that have a (large) power of two as a divisor of the dimension. In this thesis, some of the popular algorithms will be compared and combined to test the performance of several extensions of the BSP model.

3.1 3D Matrix multiplication

We will start with the BSP algorithm for the three-dimensional approach to matrix multiplication [Aga+95] to compute $\mathbf{C} = \mathbf{AB}$, where $\mathbf{A}$ is an $n \times m$ matrix, $\mathbf{B}$ is an $m \times k$ matrix and $\mathbf{C}$ is an $n \times k$ matrix. The algorithm labels the processors with a three-dimensional label $P(s,t,u)$ from a $q_0 \times q_1 \times q_2$ processor cube, so we have $p = q_0 \cdot q_1 \cdot q_2$ processors. For simplicity, we will assume $q_0 = q_1 = q_2 = q$, thus $p = q^3$. We also assume that q divides all of n, m, k. We then divide the output matrix $\mathbf{C}$ into $q \times q$ blocks $\mathbf{C}_{st}$ of size $\frac{n}{q} \times \frac{k}{q}$, input matrix $\mathbf{A}$ into $q \times q$ blocks $\mathbf{A}_{su}$ of size $\frac{n}{q} \times \frac{m}{q}$, and input matrix $\mathbf{B}$ into $q \times q$ blocks of size $\frac{m}{q} \times \frac{k}{q}$. Each of these blocks is then divided over another q processors. The distribution here is not that important, as we will be replicating the entire blocks $\mathbf{A}_{su}, \mathbf{B}_{ut}$ of the matrices. The ownership of any element (i,j) of the matrices can then be described by distribution functions.

Definition 1 (Distribution for the 3D matrix multiplication).

Let $p = q^3, q \in \mathbb{N}$ be the number of processors, labelled (s,t,u) with $0 \leq s,t,u < q$. Let $b(i) := \lceil i/q \rceil, b_n = b(n), b_m = b(m), b_k = b(k)$, where $b, m, k \in \mathbb{N}$, and $N = \{0, ..., n-1\}, M = \{0, ..., m-1\}, K = \{0, ..., k-1\}$. The matrices that need to be distributed are $\mathbf{A}, \mathbf{B}, \mathbf{C}$, with dimensions

$n \times m, m \times k, n \times k$ respectively. The distribution functions for the elements of the matrices are defined as

$$\begin{aligned}
\phi_A(i,j) &:= (\phi_A^{(s)}(i,j), \phi_A^{(t)}(i,j), \phi_A^{(u)}(i,j)) &&= \left(\left\lfloor \frac{i}{b_n} \right\rfloor, \phi_A^{(t)}, \left\lfloor \frac{j}{b_m} \right\rfloor\right), \\
\phi_B(i,j) &:= (\phi_B^{(s)}(i,j), \phi_B^{(t)}(i,j), \phi_B^{(u)}(i,j)) &&= \left(\phi_B^{(s)}, \left\lfloor \frac{j}{b_k} \right\rfloor, \left\lfloor \frac{i}{b_m} \right\rfloor\right), \\
\phi_C(i,j) &:= (\phi_C^{(s)}(i,j), \phi_C^{(t)}(i,j), \phi_C^{(u)}(i,j)) &&= \left(\left\lfloor \frac{i}{b_n} \right\rfloor, \left\lfloor \frac{j}{b_k} \right\rfloor, \phi_C^{(u)}\right).
\end{aligned}$$

Note that the distributions $\phi_A^{(t)}, \phi_B^{(s)}, \phi_C^{(u)}$ are left implicit; this is because any balanced distribution of the blocks $\mathbf{A}_{su}, \mathbf{B}_{ut}, \mathbf{C}_{st}$ gives us a balanced communication cost. We can for example choose the element-wise cyclic or block distributions, or row/column wise cyclic or block distributions, to name a few. It looks like (i,j) are swapped in the definition of ϕ_B; this is actually because ownership of a block row of $\mathbf{C}$ determines ownership of a block row of $\mathbf{A}$, and ownership of a block column of $\mathbf{C}$ determines ownership of a block column of $\mathbf{B}$.

We then use the property that $\mathbf{C}_{st} = \sum_{u=0}^{k-1} \mathbf{A}_{su}\mathbf{B}_{ut}$, for $0 \leq s < n, 0 \leq t < k$. Now, instead of computing this sum of matrix products directly on a single processor, we compute $\widetilde{\mathbf{C}}_{stu} = \mathbf{A}_{su}\mathbf{B}_{ut}$ on $P(s,t,u)$. In order to compute this product, we need to replicate $\mathbf{A}_{su}$ over $P(s,*,u)$ and $\mathbf{B}_{ut}$ over $P(*,u,t)$, by broadcasting the part of $\mathbf{A}$ owned by $P(s,t,u)$ to all $P(s,*,u)$ and the part of $\mathbf{B}$ owned by $P(s,t,u)$ to all $P(*,t,u)$. We then have al the input data to compute $\widetilde{\mathbf{C}}_{stu}$. After all processors computed their $\widetilde{\mathbf{C}}_{stu}$, $P(s,t,u)$ gathers the contributions to the part of $\mathbf{C}$ that it owns from $P(s,t,*)$ and sums them up. This operation is called a **reduce** operation. Note that matrix $\widetilde{\mathbf{C}}_{stu}$ is a local matrix, so we need to translate a global index of $\mathbf{C}$ to a local index in order to address the corresponding elements. Since we have chosen to use the block distribution, we can simply perform a modulo operation with the corresponding block sizes. A contribution to $\mathbf{C}(i,j)$ is stored in $\widetilde{\mathbf{C}}_{stu}(i \mod b_n, j \mod b_k)$. The BSP algorithm is given in Algorithm 3.1. The cost of each superstep is shown in Table 3.1. The $2l$ in superstep (2) are due to the fact that we actually merged a computation and communication superstep with the reduce operation. By summing up the cost of the supersteps and substituting the approximation $\frac{q-1}{q^3} \approx \frac{1}{q^2}$, the total cost becomes

$$T_{total} \approx \left(2\frac{nmk}{p} + \frac{nk}{q^2}\right) + \frac{1}{q^2}(nm + mk + nk)\, g + 4l.$$

If we have square matrices with $m = k = n$, this further simplifies to $T_{total} \approx \left(2\frac{n^3}{p} + \frac{n^2}{q^2}\right) + 3\left(\frac{n^2}{q^2}\right) g + 4l$.

superstep	cost
(0)	$(q-1)\cdot\left(\frac{nm}{q^3}+\frac{mk}{q^3}\right)g+l$
(1)	$2\frac{nmk}{q^3}+l$
(2)	$(q-1)\frac{nk}{q^3}(g+1)+2l$

Table 3.1: The BSP cost for Algorithm 3.1

Algorithm 3.1. A three-dimensional approach to matrix multiplication.

input : $\mathbf{A} : n \times m$ matrix, distr $(\mathbf{A}) = \phi_A$,
$\mathbf{B} : m \times k$ matrix, distr $(\mathbf{B}) = \phi_B$,
output : $\mathbf{C} : n \times k$ matrix, $\mathbf{C} = \mathbf{AB}$, distr $(\mathbf{C}) = \phi_C$.
(refer to Definition 1 for distributions.)

{Replicate $\mathbf{A}_{su}$ over $P(s,*,u)$} ▷ superstep (0)
for all $(i,j) \in N \times M \ \wedge \ \phi_A(i,j) = (s,t,u)$ **do**
put $\mathbf{A}_{i,j}$ in $P(s,*,u)$

{Replicate $\mathbf{B}_{ut}$ over $P(*,t,u)$}
for all $(i,j) \in M \times K \ \wedge \ \phi_B(i,j) = (s,t,u)$ **do**
put $\mathbf{B}_{i,j}$ in $P(*,t,u)$

{Compute the contribution to $\mathbf{C}$ of $P(s,t,u)$} ▷ superstep (1)
compute $\widetilde{\mathbf{C}}_{stu} = \mathbf{A}_{su}\mathbf{B}_{ut}$

{Sum the contributions to $\mathbf{C}$ of $P(s,t,*)$} ▷ superstep (2)
for all $(i,j) \in N \times K \ \wedge \ \phi_C(i,j) = (s,t,u)$ **do**
$C(i,j) = \text{Reduce}_+(\widetilde{\mathbf{C}}(i \mod b_n, j \mod b_k), P(s,t,*))$

4 Extending the BSP model

4.1 Generalisations in the BSP model

Now that we have gained some familiarity with what the BSP model is and how it is used in constructing and analysing algorithms, we can analyse where an extended model could further benefit by exploiting more properties of the system, as well as providing a more realistic analysis of performance.

4.1.1 Shared memory

The BSP model at its core does not account for the existence of shared memory at some layers of the architecture. However, it may be beneficial to both performance and the readability of an algorithm to utilise it, for example to store input parameters. We usually specify input and output parameters as distributed over the participating processors, which need to communicate in order to access input parameters that they do not own themselves. If we allow the input to be stored entirely in this shared memory, the processors connected to this shared memory do not have to communicate in order to read input that they would not own in the distributed setting.

4.1.2 Architectural layers in communication

In order for a parallel model to adapt to modern supercomputers, and utilise the full extent of their performance, we cannot ignore the existence of architectural layers in communication. The BSP model would still accurately model the performance when communication volume is nearly uniform between all processors, because in that case you are limited by the worst-case communication times. This is also the model parameter that we would measure in a benchmark of full h-relations. A full h-relation is a communication superstep where every processor both sends and receives exactly h data words. If somehow you are (un)lucky enough to have a communication pattern that fits better with the architectural layers, you may have a performance increase that is not explicable by the cost analysis of the BSP model.

Modern supercomputers in general consist of many nodes, each equipped with a multi-core CPU. These cores have access to a shared memory, resulting in a communication cost much lower than that of the communication network. The nodes are connected through a communication network, and the cores within each node share memory with each other. Due to this clearly hierarchical structure, it makes sense to account for at least two layers in the algorithm: a layer that models the communication between nodes, and a layer that models the computation and the communication of the cores within each of the nodes.

There are still a few ways we can further extend the hierarchy. For example, there are nodes with multiple sockets. Each of the sockets contains a multi-core CPU. While a multi-socket node provides shared memory to all cores, there is usually one of the sockets appointed as the owner of part of that memory. The owner is often appointed in the "first-touch" manner: the socket from which the memory is "touched first" (allocated) will be the owner of the memory. This means access from a core on a different socket to this memory is more expensive. In this setting, we may want to introduce another level to the hierarchy by identifying sockets as high level structure, and cores as embedded in the socket, similar to how we viewed nodes.

We can also look in the other direction; nodes can exist in the same server rack, or in the same island consisting of multiple server racks, with faster inter-connections. These nodes generally do not share memory, but only a communication network.

The general phenomenon that we will see if we generalise to the BSP model, is that we assume uniform communication between nodes, while this communication is non-uniform if we look at the pairwise communication cost, and the measured communication cost parameter is dominated by the worst-case pairwise communication cost.

4.1.3 Heterogeneous computation nodes

A subject that is a bit more exotic to the BSP model is heterogeneous computation nodes: nodes that are of different computational speed. As a supercomputer gradually gets extended with newer processing nodes, the newer nodes are slightly faster than the previous generation each time. The queue manager usually makes sure we are allocated a homogeneous set of nodes, but this is not necessarily the case. In this case, we need to normalise the cost with the computation rate of the different nodes.

Co-Processors and Accelerators

Another architectural feature that is not yet incorporated in the BSP model, is the existence of Co-Processors and Accelerators. They are both a special type of multi-processor, aimed at a specific type of computations. As the name suggests, they coexist with the regular CPU. They are often much more efficient compared to the CPU. For example, the GPU can

be used to speedup linear algebra computations. We can view the CPU and the Co-Processor or Accelerator as heterogeneous nodes. However, as they are suitable for/specialised at a certain type of computation, we need to take care that they are used for the job they are best suited for.

4.2 Towards a flexible generalisation

As we have observed, the physical architecture does not always agree with the model of the architecture, except in the most general cases. Therefore, we want to have a model that provides both (easy) analysis of the expected cost and speedup of algorithms, as well as flexible ways to extend the model for new and exotic architectures. We cannot possibly analyse the cost of algorithms for all future architectures without generalisation. We need concrete assumptions about the communication network in order to formalise the communication. However, we can leave some room for extension of the model.

4.2.1 Terminology

In order to abstract away from the physical components of a supercomputer, we introduce some basic terminology to address the different components. A **node** is a unit participating at a certain level of the hierarchy. If there are multiple levels in the hierarchy, a node at a higher level consists of a disjoint subset of nodes from the level directly below. At the leaves of the hierarchy, the components we generally call **processors** in the BSP context are also regarded as nodes for the communication.

To address the hierarchy, we use the nomenclature of the genetic relationships: the leaf nodes may have a parent , a grandparent, and further ancestors. A node also has siblings from the same parent, and we will call siblings of the direct parent uncle for short. There are a few ways how siblings can share information with each other and their cousins. We can generally assume siblings have a way to share information with each other, either through a communication network, or through shared memory. To share information with their cousins, siblings can share memory that is accessible by the parent. The parent can then communicate information from this memory with its siblings, to get information to the cousins of their children. In case the siblings do not share memory, there are two possibilities: there is one specific child, the elder child, that communicates with the parent, or there is a (simulated) communication network between cousins.

4.2.2 Separating divide from conquer

One of the ways we can parallelise an algorithm, is the divide-and-conquer paradigm: we “divide” the work over processors, and “conquer” (compute) the partial result on each processor, and aggregate such that every processor has the result it needs for further computation. We then usually

write out the entire algorithm as a sequence of communication (divide) and computation (conquer) supersteps. In general, there is also a part of the algorithm that describes the collection and aggregation of data, but this part would fall under the described "divide" part of the algorithm, as it consists of communication.

In order to allow for representation of the architectural layers, we can separate the algorithm into two parts: a divide algorithm, the communication between nodes at a certain layer, and a conquer algorithm, the computation within the nodes. The conquer algorithm can then be further specified for different node types. The conquer algorithm can also recursively consist of divide and conquer algorithms if the node has children in the hierarchy. The main environment will be denoted by E_0, and a list of sub-environments is also provided as input to the (mathematical) algorithm as $[E_1, E...]$, a list of dynamic length.

Note that this separation does not have to be strict: take for example a summation of a list of values, where only one node has the entire list. Note that this is not the best starting point for the algorithm, as the communication would certainly be more expensive than sequential computation in this case, but this example is just to better understand this statement. The "divide" part of the algorithm would be dividing the list over all active processors, and collecting the intermediate results on every node. The conquer parts of the algorithm would consist of the summation of the local part of the list, and the summation of intermediate results. However, if a node is also capable of computation (besides the computational capability in the sub-environment), and the size of the intermediate results is moderate as compared to the total input size, then it may be beneficial to just compute it sequentially on the node. As a general guideline, we can first write out the flat BSP algorithm in the original model, identify the most expensive parts, and substitute these parts by referring to a "conquer" algorithm that is dependent on the type of the node.

4.2.3 Subset synchronisation

Another useful feature that was even discussed in the initial model [Val90], but was never formalised because there is no general consensus on what the "correct" properties should be, is subset synchronisation. NestStep [Keß00] is aimed at providing subset synchronisation, mentioning some of the advantages of providing it in the model. When all processors synchronise at the same time, there will be more congestion in the network. Even if processors do not communicate data for the algorithm with each other, they have to agree not to exchange data, which is communication in itself. By supporting (nested) subset synchronisation, the network hierarchy can be reproduced, leading to less congestion and thus lower communication and synchronisation cost.

5 Benchmarking for cost prediction and reduction

In order to predict the performance and the speedup of our algorithm on a physical machine, we need to determine the machine parameters. With these parameters, we can choose the optimal configuration to run our algorithm on, given the parametrised cost of the algorithm. For a non-hierarchical BSP computer, this boils down to balancing computation, communication and synchronisation. Using too many processors leads to communication being the bottleneck. For a hierarchical architecture, we need to fill in the parameters for each level of the hierarchy, and possibly normalise the cost if the computation rate is different in the measurement of different parts of the hierarchy.

These parameters depend on the machine that will be used, and the number of processors that is used. In the case of a supercomputer, the parameters may even change each execution if nodes are allocated differently. The parameters are also dependent on the library that is used for communication and synchronisation. In order to get a representative measurement of the parameters, we need a portable benchmarking tool.

5.1 BSP Benchmarking

A framework for BSP benchmarking of a non-hierarchical machine is given by Bisseling [Bis04]. First, the computation rate r is measured using DAXPY[1] operations of increasing length n. DAXPY operations are of the form $\vec{y} = \alpha\vec{x} + \vec{y}$. To get a more representative measurement, we interleave them with the similar operation $\vec{z} = \vec{z} - \beta\vec{x}$. Each DAXPY pair is then $4n$ floating-point operations (flops), and $O(n)$ memory operations. Any algorithm needs memory access, so we generalise this to be the cost of performing $4n$ flops. We measure the time for values n that are powers of 2, and perform a few repetitions before measuring the time, in order to negate the cost of retrieving the elapsed time, and reduce the impact of timer resolution. We report the minimum, maximum and average flop rate of the participating processors, in order to spot imbalances, and we take the average flop rate of the maximum n to be the representative of our computation rate r.

[1] Double Precision A times X Plus Y.

What remains is to compute the communication cost parameter g and the synchronisation cost parameter l. We will measure them by performing full h-relations. An h-relation, as mentioned before, is a communication superstep in which any processor receives or sends *at most* h data words. The BSP model assumes the cost to be linear in g, so the cost of such a superstep would be $T_{\text{comm}}(h) = hg + l$. A full h-relation is a communication superstep in which *every* processor receives *exactly* h data words. We achieve this by performing communication in a cyclic fashion. If we would have perfect measurements, it would be sufficient to perform two measurements for two distinct values of h. This would then be all the information we needed to derive the parameters. However, as measurements are never perfect, it is more robust to measure the cost of the communication for a range of values $\{h_0, ..., h_1\}$, and perform a linear regression by a least-squares approximation.

Lemma 1.

Let $t(h)$ be the measured times for full h-relations with $h \in \{h_0, h_0 + 1, ..., h_1\}$. A least-squares approximation minimises the sum of the squares of the error

$$S(g,l) = \sum_{h=h_0}^{h_1} (t(h) - T_{\text{comm}}(h))^2 = \sum_h (t(h) - (hg + l))^2,$$

with the approximation of the parameters

$$g = \frac{\sum_h t(h) - \alpha \sum_h ht(h)}{\sum_h h - \alpha \sum_h h^2}, \tag{5.1}$$

$$l = \frac{\sum_h ht(h) - g \sum_h h^2}{\sum_h h}, \tag{5.2}$$

where

$$\alpha = \frac{h_1 - h_0}{\sum_{h=h_0}^{h_1} h}.$$

Proof. If we regard $\vec{t} = t(h), \vec{T} = T_{\text{comm}}(h)$ as vectors, then

$$S(g,l) = \sum_h (t(h) - T_{\text{comm}}(h)) = \|t - T\|_2^2$$

is a monic quadratic expression, so the only global extremum is a global minimum, and it is attained for $\frac{\partial S}{\partial g} = \frac{\partial S}{\partial l} = 0$. We have

$$\frac{\partial S}{\partial g} = -2 \sum_h (t(h) - (hg + l)) \cdot h = 2 \left(g \sum_h h^2 + l \sum_h h - \sum_h ht(h) \right),$$

$$\frac{\partial S}{\partial l} = -2 \sum_h (ht(h) - (hg + l)) \cdot 1 = 2 \left(g \sum_h h + \sum_h l - \sum_h t(h) \right).$$

Solving $\frac{\partial S}{\partial g} = 0$ for l, and substituting l and α into $\frac{\partial S}{\partial l} = 0$, and solving for g gives us Equation (5.1) and Equation (5.2). □

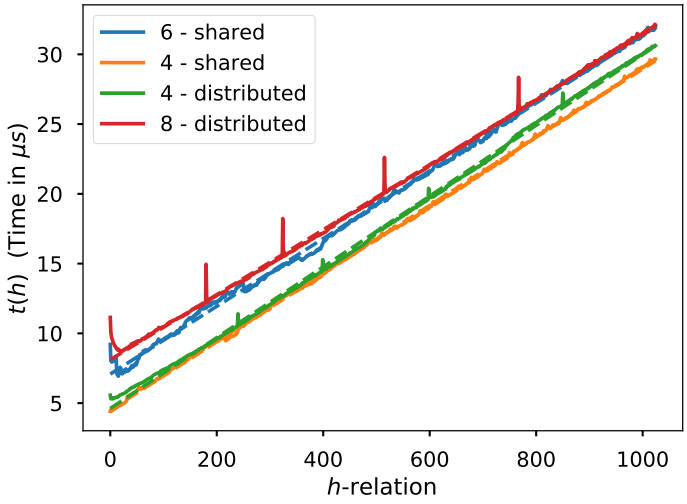


Figure 5.1: Benchmark of linearly increasing full h-relations, using BSPedupack [Bis04] and MulticoreBSP version 1.2.0 [Yze14]. The benchmark is performed for different values of p, both on shared memory within the node, and distributed memory over nodes.

The result of such a benchmark is shown in Figure 5.1. As we can see, this corresponds very well to our expectation of a linear relation. As we increase the number of participants, the noise in this benchmark also increases. We may want to choose a larger package size, to further reduce the noise in the timings. If we choose package size b, then we need to fill in $\widetilde{h} = bh$ in the terms of the sums in the approximation.

Corollary 1.

Let $t(bh)$ be the time measured for full h-relations with packets of size b, with $h \in \{h_0, h_0 + 1, ..., h_1\}$. The least squares approximation is then given by

$$\alpha = \frac{h_1 - h_0}{b \sum_h h}, \tag{5.3}$$

$$g = \frac{\sum_h t(bh) - \alpha \cdot \sum_h ht(bh)}{b \sum_h h - \alpha \cdot b \sum_h h^2}, \tag{5.4}$$

$$l = \frac{\sum_h t(bh) - gb \sum_h h^2}{\sum_h h}. \tag{5.5}$$

We can now further simplify the $\sum_h h, \sum_h h^2$ terms by their closed form representation

$$\sum_{h=h_0}^{h_1} h = \frac{h_1(h_1+1) - (h_0-1)h_0}{2},$$

$$\sum_{h=h_0}^{h_1} h^2 = \frac{h_1(h_1+1)(2h_1+1) - (h_0-1)h_0(2h_0-1)}{6}.$$

This benchmark can be used to measure the parameters of a set of processors that is deemed to have a uniform communication cost and homogeneous computation rate. Some measurements of r, g, l are shown in Figure 5.2, using MulticoreBSP [Yze14] and the BSPedupack [Bis04] benchmark, both on a single node, and distributed over nodes with a single processor per node. What is interesting to see is that g (up to some artefacts) seems to grow linearly with the number of participants, and is not that different for single-node communication vs. multi-node communication. That being said, the linear growth in g is moderate compared to the growth in l. For a single node, l also seems to grow linearly, while for multiple nodes, it seems to grow superlinearly. This superlinear growth in measured values of l is also mentioned by Bisseling [Bis04].

5.2 Benchmarking Hierarchical Machines

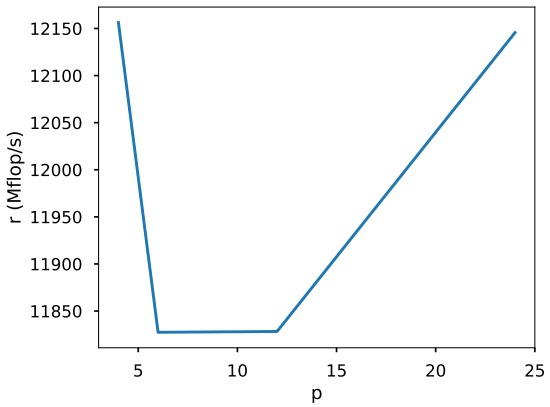
In case we have a hierarchical machine, the cost of communication is not uniform, and the parameters measured by the aforementioned benchmarking method will be dominated by the worst-case pairwise costs. Synchronisation in subsets will surely not increase the parameter g, it may well decrease it depending on whether the subset configuration will exclude or include the worst-case pairwise costs. As we noticed in Figure 5.2, l grows (super)linearly with the number of participants, so this is another reason to split into subsets.

For these reasons, subset synchronisation, where possible, always seems like a good idea, but can we do better? As the communication cost is dominated by the worst-case pairwise cost, we can try to measure the pairwise communication cost, and relabel the nodes such that the nodes that exhibit this worst-case cost are not grouped together in a subset. However, it can also be the case that bandwidth is uniform, but latency is dominating. In this case we want partition the nodes into sets such that the intra-node latency within the subset excludes the high-latency connections.

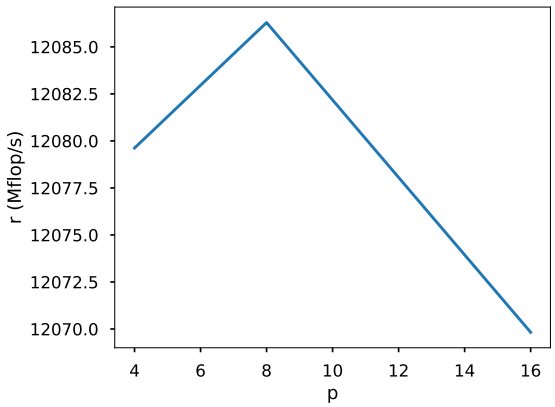
5.2.1 Benchmarking Pairwise Interaction

In any case, we need pairwise measurements of communication times. Such a measurement is only possible if the architecture supports it, but this is the case for most modern supercomputers. We will also need a library that supports pairwise communication. For the purpose of this benchmark, we can use MPI primitives, as they support pairwise communication.

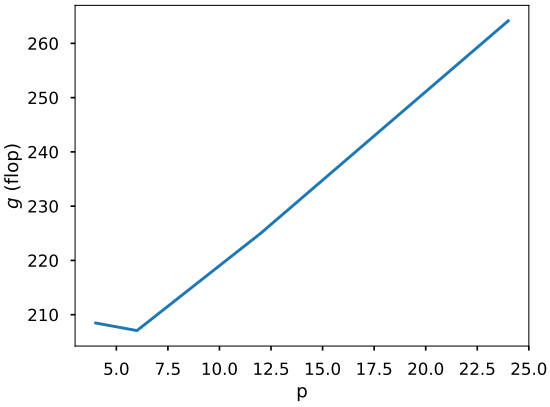
For this purpose, we need a function $f_i(s)$ that maps to the ping-pong partner of s in round i. We will refer to this as the **ping-pong partner function**. Let p be the number of nodes, with label $s \in P = \{0, 1, ..., p-1\}$. In order to perform these benchmarks, we need a mapping



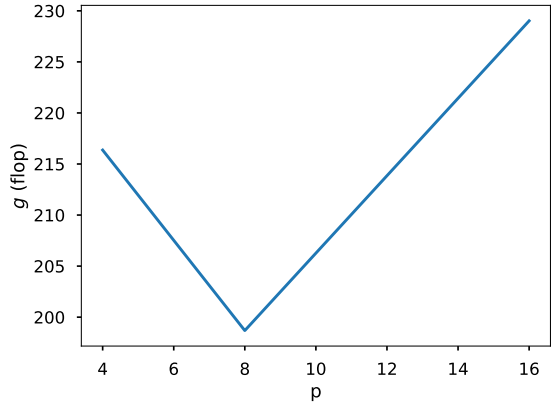
(a) Measurement of r on a single node.



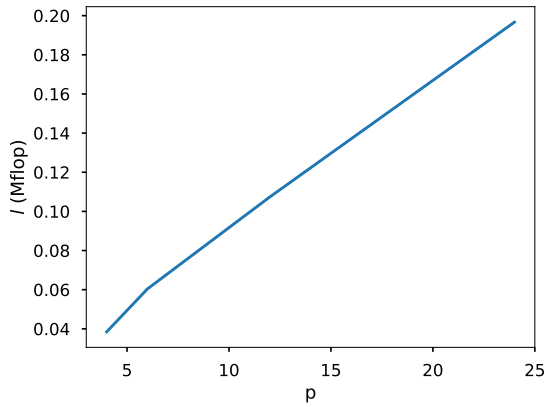
(b) Measurement of r with p nodes, each with only one processor.



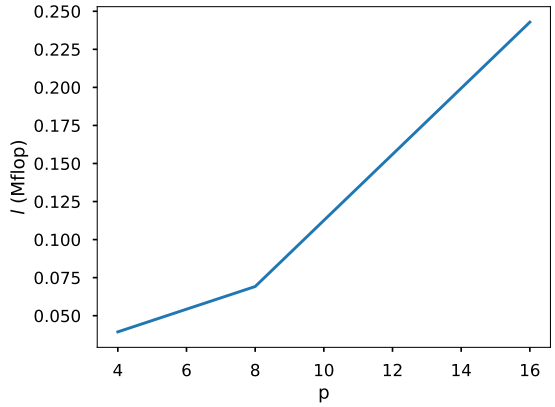
(c) Measurement of g on a single node.



(d) Measurement of g with p nodes, each with only one processor.



(e) Measurement of l on a single node.



(f) Measurement of l with p nodes, each with only one processor.

Figure 5.2: Measurement of several values of r, g, l for different p, using MulticoreBSP [Yze14] and BSPedupack [Bis04]. Both g, l are normalised from the measured r to $r_0 = 8528.27$, in favor of comparison later on.

$f_i : [0, ..., p-1] \mapsto [0, ..., p-1]$ for each round $i \in Q = \{0, 1, ..., q-1\}$ that satisfies the following properties.

i) For each $i \in Q$ and $s \in P$: $f_i(s) = t \iff f_i(t) = s$.

ii) For each $i \in Q$: $\{f_i(s) | s \in P\} = P$.

iii) For each $s \in P$: $\{f_i(s) | i \in Q\} = P$.

Property i) means that in each round, communication can only happen pairwise, or with the node itself. Property ii) means that in each round, all nodes have a source from which they receive data. Property iii) means that each node communicates with every other node at least once. Note that because of iii), we need $q \geq p$.

Definition 2 (Exclusive-OR (XOR)).

For a single bit, the Exclusive-OR (XOR) operation is equal to 1 if either one of the bits is equal to 1, but not both. Otherwise, it is equal to 0, so

$$b \text{ XOR } c := \begin{cases} 0 & \text{if } b = c, \\ 1 & \text{if } b \neq c. \end{cases}$$

Note that $b = c$ implies that either $b = c = 1$ or $b = c = 0$, and $b \neq c$ implies that either one is equal to 1, but not both.

The bitwise XOR (also called XOR for short) is the extension of this operation to integers: it acts on the binary representation, and applies the XOR operation to the bits of x and y. Let $x = x_n...x_2x_1x_0$ and $y = y_n...y_2y_1y_0$ be the binary representation of the integers x, y. Then $z = x \text{ XOR } y := z_n...z_2z_1z_0$, with $z_i := x_i \text{ XOR } y_i$. The identity element for bitwise XOR is 0. The bitwise XOR is **associative** and **commutative**, as well as its own **inverse**, so $(x \text{ XOR } y) \text{ XOR } y = x$. The latter follows easily from the associativity, and $y \text{ XOR } y = 0$ being the identity element.

With the definition of the XOR operation, a possible candidate for ping-pong partner function is

$$f_i(s) = \begin{cases} s \text{ XOR } i & \text{if } s \text{ XOR } i < p, \\ s & \text{otherwise ,} \end{cases} \tag{5.6}$$

with $q = 2^{\lceil \log_2 p \rceil}$. When p is a power of 2, this is the optimal number of rounds, but this can asymptotically have a redundancy factor 2. The worst case is when $p = 2^k + 1$. Then we need $2^k - 1$ additional rounds, and in these rounds, only the node with label $s = 2^k$ will communicate in sequence with all other nodes. For the purpose of the benchmark, this suffices. Properties i), ii), iii) can easily be verified for this function: either $f_i(f_i(s)) = (s \text{ XOR } i) \text{ XOR } i = s \text{ XOR } (i \text{ XOR } i) = s$ if $s \text{ XOR } i < p$

or $f_i(f_i(s)) = f_i(s) = s$ if s XOR $i \geq p$. Thus f_i satisfies property i). Properties ii) can be proven by contradiction: suppose $\exists s, t \in P, s \neq t$ such that $f_i(s) = u, f_i(t) = u$. Then $f_i(u) = f_i(f_i(s)) = s$ by property i), but also $f_i(u) = f_i(f_i(t)) = t$, so then it must hold that $t = s$, contradicting our assumption. Thus, $f_i(s)$ is unique for all $s \in P$, *and* by definition, $0 \leq f_i(s) < p$, so we have p unique values for $0 \leq s < p$, implying property ii) holds. Similarly, for property iii), s XOR i takes q unique values for $0 \leq i < q$. Of those values, there are $q - p$ values larger than p, leaving p unique values and $q - p$ values equal to s. The union of these values must then be equal to P.

If we assume that no action will be taken if $f_i(s) = s$ (or at least not through the network), then this means that the later rounds will have less congestion on the network. To overcome this imbalance, we can introduce some randomisable parameters into the function. A candidate for this is

$$\widetilde{f}(i, s, \alpha, \beta) = (((s + \beta) \text{ XOR } \alpha) \text{ XOR } i) - \beta \mod q, \tag{5.7}$$

$$f_i(s, \alpha, \beta) = \begin{cases} \widetilde{f}(i, s, \alpha, \beta) & \text{if } \widetilde{f}(i, s, \alpha, \beta) < p, \\ s & \text{otherwise ,} \end{cases} \tag{5.8}$$

with random $\alpha, \beta \in \{0, 1, ..., q - 1\}$. It now suffices to show that $f_i(f_i(s)) = s$, as all properties i), ii), iii) follow from this property. The case $\widetilde{f}(i, s, \alpha, \beta) \geq p$ trivially satisfies this property, so what remains to be proven is that $\widetilde{f}(i, \widetilde{f}(i, s, \alpha, \beta), \alpha, \beta) = s$. We have that $x - \beta \mod q + \beta = x \mod q$. Since q is a power of 2, and $i, \alpha < q$, a XOR operation of $y \in Q$ with either i or α will never change a bit of y such that y XOR $i > q$ or y XOR $\alpha > q$, thus

$$y \text{ XOR } \alpha \text{ XOR } i \mod q = y \mod q \text{ XOR } \alpha \text{ XOR } i.$$

Then it follows that

$$(y \text{ XOR } \alpha \text{ XOR } i \mod q) \text{ XOR } \alpha \text{ XOR } i = y \mod q.$$

Putting it all together, we have

$$\begin{aligned} \widetilde{f}(i, \widetilde{f}(i, s, \alpha, \beta), \alpha, \beta) &= \Big(((((s + \beta) \text{ XOR } \alpha) \text{ XOR } i) - \beta \mod q + \beta) \\ &\qquad \text{XOR } \alpha) \text{ XOR } i\Big) - \beta \mod q \\ &= \Big(((((s + \beta) \mod q \text{ XOR } \alpha) \text{ XOR } i)) \\ &\qquad \text{XOR } \alpha) \text{ XOR } i\Big) - \beta \mod q \\ &= ((s + \beta) \mod q) - \beta \mod q \\ &= s \mod q = s \end{aligned}$$

Now instead of just performing repetitions of the communication for more stable measurement, we can also perform multiple iterations over Q, with a different (α, β) pair for each full iteration. Note that if we were to

use only α (equivalent to $\beta = 0$ fixed), then we only reorder rounds, but we do not change the internal structure of the rounds. Instead we could take for example $i = \beta = 1$, and let $\alpha = 0$ for the moment. Let s even, then $s + \beta$ XOR $i = s_n...s_2s_11$ XOR $0...001 = s_n...s_2s_10 = s$, thus $(s + \beta)$ XOR $i - \beta = s - 1$, while $(s + 1 + \beta)$ XOR $i - \beta = s + 1$. With $\beta = 0$, this would be the other way around, thus we change the internal structure of the rounds by choosing different β values. By both randomising α, β, we randomise the order and the structure of rounds.

Benchmarking pairwise bandwidth

Because g is dependent on the bandwidth, it can be interesting to try to measure this bandwidth in this pairwise fashion, because we can then try to split the low-bandwidth pairs into different node subsets. We can perform a similar benchmark to the benchmark of Section 5.1: a sequence of pairwise full h-relations for each pair $(s, f_i(s))$, for each round $i \in Q$. Then we can get a measurement of g for the pair from the least squares approximation. In case of large differences in the value of g, this means bandwidth for those pairs is limited and they will be dominating the cost of the subset. In Figure 5.3, we can see that there is certainly non-uniformity in bandwidth. This could either be due to congestion, or more likely because the nodes are dynamically allocated, and nodes that are closer have higher bandwidth. The latter seems more likely, as there exist blocks around the diagonal where communication cost is lower, thus bandwidth is higher.

Benchmarking pairwise latency

We could retrieve the latency parameter from the same benchmark as above, but then the measured latency could be highly dependent on the noise in the bandwidth. This is shown in Figure 5.4. We see similar patterns occurring, so the noise does not distort these patterns. This indicates that optimising either bandwidth or latency would optimise

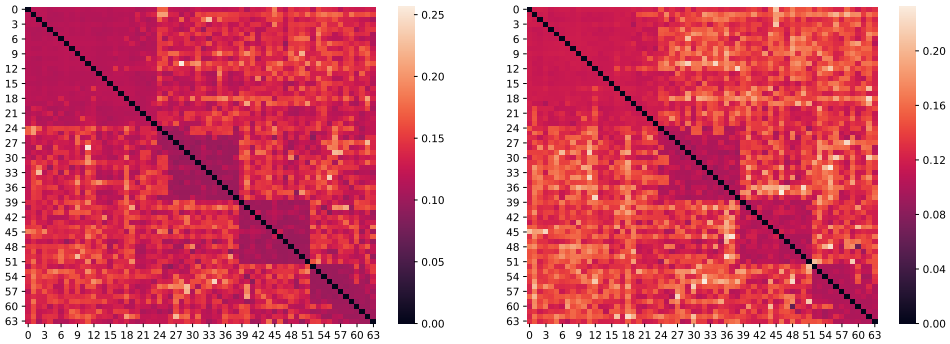


Figure 5.3: Pairwise measurement of bandwidth, for 2 separate runs with 64 physical nodes in which the main process communicates with other nodes.

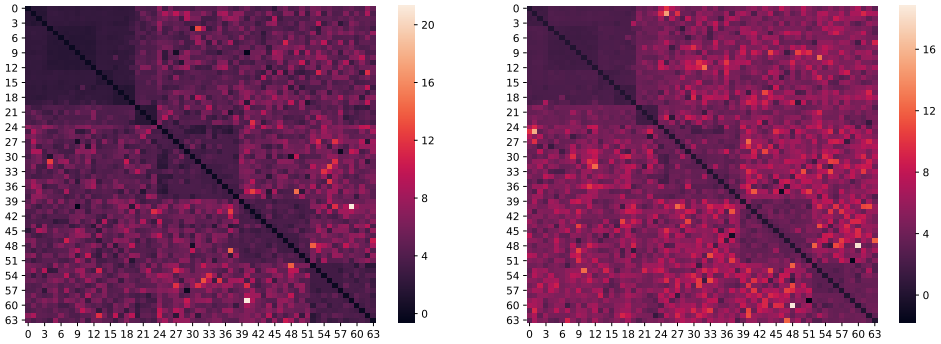


Figure 5.4: Pairwise measurement of latency, for 2 separate runs with 64 physical nodes in which the main process communicates with other nodes.

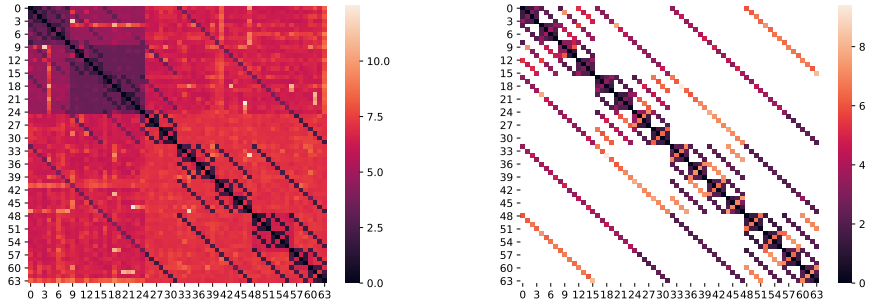
both. Note however that this is machine dependent. Machines with similar communication networks will show similar patterns, but different communication media may impact the similarity. For an indication of the startup latency ℓ, it would be more useful to measure the single-word communication time. The startup latency is then an indication of the physical distance of the nodes, both in number of hops internally in the network, and the length of the connecting medium in case this affects the latency. In case of hops in the network, this could be an indication of congestion points that we want to avoid. An example of the startup latency can be viewed in Figure 5.5. We can see clear patterns of pieces of diagonals in the startup latency, which means we can measure some of the physical properties of the system. There are however also a lot of other patterns that may either be due to the small communication size, or because of network congestion. If we were to perform the 3D matrix multiplication with a $4 \times 4 \times 4$ processor cube, then Figure 5.5 (b) shows the weights and pattern involved in the communication. As we can see, this closely corresponds to the physical connections present in the machine, thus there will be little to improve, certainly if the other patterns only occur due to congestion.

5.2.2 Relabelling nodes

Now that we have a measurement of the communication parameters, we want to relabel nodes such that either the minimum bandwidth in the subset that it is a part of is maximal (so the maximum communication cost is minimal), or the average latency within the subset is minimal. Of course, the subsets are dependent on the data distribution.

Hierarchical communication patterns

In case we have a hierarchical communication pattern, we can recursively apply the multiple-way Kernighan-Lin [KL70] graph-partitioning in a top-down fashion, to determine the optimal configuration of the nodes.



(a) Startup latency.

(b) Weights involved in a communication pattern with a $4 \times 4 \times 4$ processor cube.

Figure 5.5: Pairwise measurement of startup latency, run with 64 physical nodes in which the main process communicates with other nodes.

The Kernighan-Lin algorithm minimises the weight of the edges that cross between partitions. Instead of minimising the inter-partition weight, we want to minimise the intra-partition weight. We can modify the Kernighan-Lin algorithm to work on the inverse of the cost, thus maximising the inter-partition weight, and thus leaving out high cost edges from the intra-partition connections. The algorithm is shown in Algorithm 5.1. Note that we need to modify the cost as mentioned before applying the algorithm. As Kernighan and Lin mentioned, the algorithm needs multiple passes over all pairs of subsets, but they experimentally found that the convergence is fast.

Multi-dimensional cartesian distribution

In case we have a multi-dimensional cartesian distribution such as the aforementioned processor cube, nodes are part of multiple subsets: one subset in each dimension. If we only needed to determine the subsets for a single dimension, a partitioning algorithm like Kernighan-Lin [KL70] for multiple-way graph partitioning would work, but since the node is part of a subset in each dimension, normal partitioning algorithms will not work, as optimising the partitioning in one dimension may adversely affect the cost in the subset in another dimension. We can however use Kernighan-Lin to determine some indication of a lower bound, by partitioning into the largest subset size present in any dimension, that is, let the desired configuration be of dimensions $q_0 \times q_1 \times ... \times q_{d-1}$. Then we can perform the Kernighan-Lin algorithm with subset size $q = \max_i q_i$ to get some indication of a lower bound. Note that due to only having the pairwise optimality guaranty from the multiple-way Kernighan-Lin algorithm, we are not sure that we are in a global minimum, but it is very hard to beat this (local) minimum once the elements are part of multiple subsets.

Algorithm 5.1. Multi-way Kernighan-Lin algorithm.

input : $\mathbf{D}$: a matrix of pairwise distances
S : a list of initial subsets, $S = \{S_0, S_1, ..., S_{k-1}\}$,
$S_i \cap S_j = \emptyset \forall 0 \leq i, j < k \wedge i \neq j$.
output : S : a list of improved subsets.

```
improved := 1
while improved = 1 do
    improved := 0
    for 0 ≤ i < k − 1 do
        for i < j < k do
            Δ := ∅
            Initialize L^(g), L^(a), L^(b) as empty lists
            while S_i\Δ ≠ ∅ ∧ S_j\Δ ≠ ∅ do
                for all a ∈ S_i\Δ do
                    E_a := Σ_{b∈S_j} D(a, b)
                    I_a := Σ_{α∈S_i} D(a, α)
                    D_a := E_a − I_a
                for all b ∈ S_j\Δ do
                    E_b := Σ_{a∈S_i} D(b, a)
                    I_b := Σ_{β∈S_j} D(b, β)
                    D_b := E_b − I_b
                Let g(a, b) = D_a + D_b − 2 · D(a, b)
                (a, b) := arg max {g(a, b)|a ∈ S_i\Δ ∧ b ∈ S_j\Δ}
                Δ := Δ ∪ {a, b}
                Append g(a, b) to L^(g)
                Append a to L^(a)
                Append b to L^(b)
            end while
            k := arg max_{0≤k<|L^(g)|} Σ_{ℓ=0}^{k} L_ℓ^(g)
            if Σ_{ℓ=0}^{k} L_ℓ^(g) > 0 then
                improved := 1
                for 0 ≤ ℓ < k do
                    Swap L_ℓ^(a) with L_ℓ^(b)
                end for
            end if
        end for
    end for
end while
```

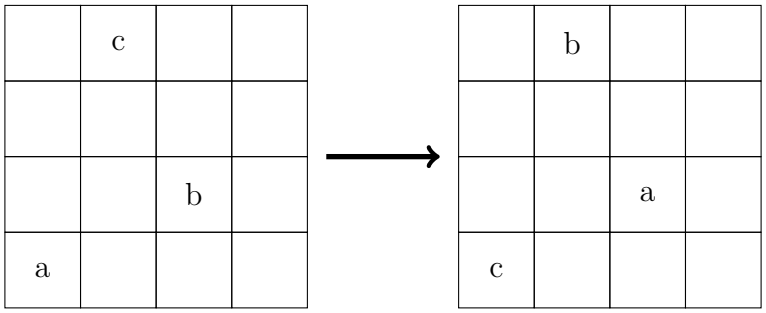


Figure 5.6: Triplet permutation in a 2-dimensional processor layout.

To determine a better configuration in this setting, we can try to perform a randomised greedy hill-climbing algorithm with simple operations. We can, for example, use random pairwise swaps, and accept the first improvement. To make sure we have considered all swaps, we can build a list of all possible swaps and randomise it. We then asses them in sequence. If we have an improvement, we rebuild this list and start again. If we have traversed the entire list without finding an improvement, we know that we are done. To allow slightly more complex changes, we can also perform a permutation over random triplets, that is, if we have the triplet (a, b, c), we permute it to (c, a, b), so c takes the label of a, a takes the label of b and b takes the label of c. The 2-dimensional case is illustrated in Figure 5.6. We can also reduce the number of candidates by choosing the first element of all the triplets, a, to be the worst-case sample, and b to be in a different subset than a. The worst-case sample is in this case the sample that is part of the worst-case subset, and has the largest total distance to all members of the subset. This ensures that the change will be effective, as the worst-case cost will mostly be dominating. As the algorithm has random components, and we have many processors at our disposal, we can perform this algorithm on each of the processors and take the best solution as the node relabelling.

Algorithm 5.2 Randomised greedy hill-climbing, triplet permutation.

input : $\mathbf{D}$: a matrix of pairwise distances
S : a list of initial subsets, $S = \{S_0, S_1, ..., S_{k-1}\}$.
output : S : a list of improved subsets.

`improved` $:= 1$
$C_{\max} := \max_{S_i \in S} \left(\sum_{s_0 \in S_i} \sum_{s_1 \in S_i} \mathbf{D}(s_0, s_1) \right)$
while `improved`$=1$ **do**
  $S_{\max} := \arg\max_{S_i \in S} \left(\sum_{s_0 \in S_i} \sum_{s_1 \in S_i} \mathbf{D}(s_0, s_1) \right)$
  $s_{\max} := \arg\max_{s_j \in S_{\max}} \left(\sum_{s_1 \in S_{\max}} \mathbf{D}(s_j, s_1) \right)$
  {Build a list of pairs such that $s_{\max}$ is moved to a different subset}
  $\Pi := \{(s_1, s_2) | s_1 \in \cup_{S_i \in S \setminus \{S_{\max}\}} \wedge s_2 \in \cup_{S_i \in S} S_i\} \setminus \{s_{\max}, s_1\}\}$
  `improved` $:= 0$
  while `improved` $= 0 \wedge |\Pi| \geq 1$ **do**

$s_1, s_2 \leftarrow (s_1, s_2) \in_{\text{random}} \Pi$
Remove (s_1, s_2) from Π
Permute $(s_{\max}, s_1, s_2) \rightarrow (s_2, s_{\max}, s_1)$
$C_1 := \max_{S_i \in S} \left(\sum_{s_0 \in S_i} \sum_{s_1 \in S_i} \mathbf{D}(s_0, s_1) \right)$
if $C_1 < C_{\max}$ **then**
    $C_{\max} := C_1$
    `improved` $:= 1$
else
    Undo permutation
end if
end while
end while

Finding the next improvement has a computational cost of $O(p^2 \cdot d \cdot \sqrt[d]{p})$: there are $O(p^2)$ candidate triplets, and for each member of the triplet, we need to update the costs in d dimensions of size $\sqrt[d]{p}$. The random part is hard to predict: the first improvement will stop the iteration, so there should be some probability factor reducing the p^2 for each iteration round, where the probability of finding an improvement should have a decreasing trend over the rounds. Note that the probability is not strictly decreasing: a change in the node labelling may increase the number of triplets that improve the overall cost in the next round. However, the number of rounds (certainly experimentally) is much larger than $O(p)$, thus this algorithm does not scale well with p. To overcome this, we can coarsen the node graph: we can put together (a small number of) nodes that are closest to each other, and recompute the distance between these subsets as the average distance between the members of the subsets. This way, the relabelling algorithm can scale much better. The subset size has to be (a divisor of) the size of one of the dimensions, as this makes sure that these nodes are in the same subset. After uncoarsening the graph, we can either accept the solution or perform more random hill-climbing, or we can choose to perform a fully greedy hill-climbing algorithm (greedily choosing the best place to move the worst-case sample to) after the uncoarsening.

6 SyncLib

In order to perform the experiments with pairwise benchmarking and relabelling nodes for better communication cost parameters, low-level control over the communication and synchronisation mechanism is needed. Therefore, the basis of a new library is laid during this thesis: SyncLib. SyncLib is an open-source library[1], with at its core a modern C++ interface with BSP-like primitives. SyncLib is also extended with other communication and synchronisation related utilities, like MPI bindings for C++, and utilities for threading outside the BSP context.

6.1 Primitives

Many of the primitives that are supported in the BSPlib standard are (or will be) supported in SyncLib. Some additional primitives will be added to support splitting and relabelling nodes in an environment. An overview of the most used primitives is found in Table 6.1.

6.1.1 Environments

The most important part of SyncLib is the notion of environments. Environments describe the (sub)set of nodes or processors that participate in a certain superstep. Environments are therefore the basis of all communication and synchronisation. Whereas BSPlib primitives operate globally for the process, SyncLib primitives will operate on an environment. By scoping primitives to an environment, we can support nesting of environments, and thus allow subset synchronisation. The environment will also be a parameter to the algorithm, so that the algorithm can be specialised to different environments.

Two main environments are implemented for the basis of SyncLib: an environment based on shared memory, and an environment based on MPI as a communication and synchronisation backend. Through MPI, SyncLib will be able to communicate between nodes on most modern multi-node supercomputers. By utilising shared memory for the intra-

[1]The library is hosted on GitHub, located at https://github.com/Zefiros-Software/SyncLib.

SyncLib	BSPlib standard
`tEnv` (environment type)	N\A

`tEnv env(argc, argv)`
`env.Resize(p)`
`env.Run(fn, arguments...)`
	`bsp_init(fn, argc, argv)`
	`bsp_begin(p)`
	`bsp_end()`

`env.Size()`	`bsp_nprocs()`
`env.Rank()`	`bsp_pid()`

`tEnv::SharedArray<int> x(env, n)`	`int *x = malloc(n * sizeof(int))`
	`bsp_push_reg(x, n * sizeof(int))`
	`bsp_pop_reg(x)`
	`free(x)`

`int y = ...`
`x.PutValue(t, y, offset)`
	`int y = ...`
	`bsp_put(t, &y, x,`
	`offset * sizeof(int),`
	`sizeof(int))`

`std::vector<int> y(m)`
`x.Put(t, y.begin(), y.end(), offset)`
	`int *y = malloc(m * sizeof(int))`
	`bsp_put(t, y, x,`
	`offset * sizeof(int),`
	`m * sizeof(int))`

Table 6.1: Relation between some example primitives of SyncLib and the BSPlib standard.

node communication between processors, there will be less congestion on the inter-node communication network, and intra-node communication will be faster than inter-node communication.

Parametrisation of environments

Using the tools the modern C++ language provides, environments will be parametrised by a template parameter. Templates can then be specialised for different types of environments. Snippet 6.1 shows an example of an algorithm parametrised by the environment. The program in Snippet 6.1 computes an approximation to

$$\sum_{n=1}^{\infty} \frac{1}{n^2},$$

known as the Basel Problem. Euler first proposed the solution to be $\frac{\pi^2}{6} \approx 1.644934$ [Eul40]. We observe that $\lim_{n\to\infty} \frac{1}{n^2} = 0$, so we can try to approximate the sum by cutting of the series; the algorithm computes the truncated sum by computing an inner product of $\vec{x} = (1/1, 1/2, ..., 1/n)^T$ with itself.

Snippet 6.1. Approximation of the solution to the Basel Problem.

```
template<typename tEnv>
double InnerProduct(tEnv &env,
```

```
                    std::vector<double> &x,
                    std::vector<double> &y)
{
    size_t p = env.Size();
    size_t s = env.Rank();
    // using shared array with put for partial inprod
    tEnv::SharedArray<double> partialInprod(env, p);

    double alpha = 0.0;

    for (size_t i = 0, iEnd = x.size(); i < iEnd; ++i)
        alpha += x[i] * y[i];

    for (size_t t = 0; t < p; ++t)
        partialInnerProducts.PutValue(t, alpha, s);

    env.Sync();

    // Sum the partial inner products
    return std::accumulate(partialInnerProducts.begin(),
                           partialInnerProducts.end(),
                           0.0);
}

template<typename tEnv>
void BaselProblem(tEnv &env, size_t n)
{
    size_t p = env.Size();
    size_t s = env.Rank();
    size_t nl = (n + p - s - 1) / p;
    std::vector<double> x(nl);

    for (size_t i = 0; i < nl; ++i)
    {
        const size_t iGlob = i * p + s;
        x[i] = 1.0 / (iGlob + 1);
    }

    SyncLib::Util::Timer<std::chrono::microseconds> timer;
    // Let everyone start at the same time
    env.Barrier();
    timer.Tic();

    double alpha = InnerProduct(env, x, x);
    // Measure the time when everyone is done
    env.Barrier();
    double elapsed = timer.Toc();

    printf("Processor␣%d:␣solution␣to␣the␣Basel␣problem␣"
           "with␣reciprocals␣up␣to␣1/%d^2␣is␣%.6f\n",
           s, n, alpha);

    if (s == 0)
        printf("This␣took␣only␣%.6lf␣microseconds.\n",
               elapsed);
}
```

```
int main(int argc, char **argv)
{
    using tEnv = SyncLib::Environment::SharedMemoryBSP;
    tEnv env(argc, argv);

    // n can be queried from the user, but we fix it
    // for the sake of the example
    size_t n = 100000;

    env.Run(BaselProblem<tEnv>, n);

    return EXIT_SUCCESS;
}
```

6.1.2 Run

The program in Snippet 6.1 already shows many of the primitives and utilities provided by SyncLib. The entrypoint of any BSP algorithm will be the `Run` primitive on the environment; it will pass the environment as the first argument to the specified algorithm, as well as any other parameters that are specified. Compared to the BSPlib standard, this replaces `bsp_init`, `bsp_begin` and `bsp_end`.

6.1.3 Size and Rank

The `Size` and `Rank` primitives, as the names suggest, retrieve the size of the environment, and the rank of the processor within the environment. They correspond to the `bsp_nprocs` and `bsp_pid` primitives respectively. The renaming corresponds more closely to the MPI naming of the concepts, and fits better with the abstraction to nodes.

6.1.4 Split and Reorder

Primitives that are not part of the BSPlib standard are `Split` and `Reorder`. SyncLib introduces the `Split` primitive to facilitate subset synchronisation, with syntax

```
auto &subEnv = env.Split(part, newRank).
```

The `part` parameter will indicate the part the processor is a member of in the subset-environment, and the `newRank` parameter is an optional parameter to reorder the processors in the subset-environment; the rank in the subset-environment is determined by sorting tuples `(newRank, oldRank)`. This way, no two processors will share the same rank in the subset-environment. Two examples of such splits are shown in Figure 6.1 and Figure 6.2. In fact, by combining these two, we can create a processor matrix of 3×3 in which only communication in rows or columns of the processor matrix occurs. Besides reducing the synchronisation (and possibly communication) time, the split of the environments also simplifies

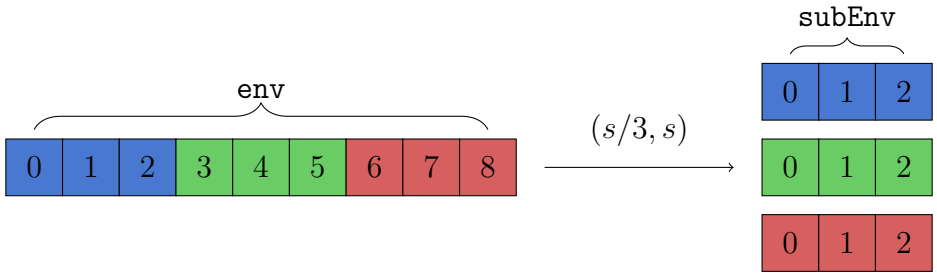


Figure 6.1: Example of splitting an environment with $p = 9$ into three equal parts `auto &subEnv = env.Split(s / 3, s )` in a blockwise fashion.

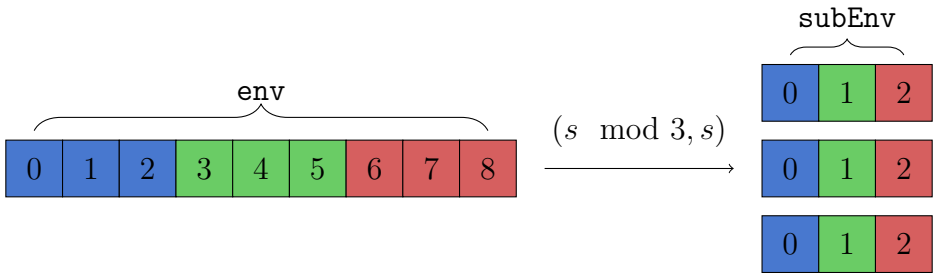


Figure 6.2: Example of splitting an environment with $p = 9$ into three equal parts `auto &subEnv = env.Split(s % 3, s )` in a cyclic fashion.

computing the target processor, as we can just address the i-th participant in the processor column, or the j-th participant in the processor row, without index transformations. While the subset synchronisation was inspired by NestStep [Keß00], the biggest difference is the dynamic coexistence of subset-environments, where multiple subset-environments can coexist in SyncLib for the entire duration of the algorithm.

The `Reorder` primitive will create a second environment

```
auto &reorderedEnv = env.Reorder(newRank),
```

in which the processors are relabelled, again in the order of the sorted tuples `(newRank, oldRank)`. Communication and synchronisation is separate in these environments, so communication that happens in the reordered environment is not synchronised upon synchronisation of the initial environment and vice versa. The reordered environment may be useful in case we have two separate logical orderings of processors, or in case we want to buffer some communication over multiple (program) supersteps.

6.1.5 Communication

Communication in SyncLib works similar to the BSPlib standard: variables are registered (but on an environment now), and you are provided with Put, Get and Send primitives.

Shared variables

There are two basic types of variables that can be shared: single values and arrays of values, with types `tEnv::SharedValue` and `tEnv::SharedArray` respectively. They both have template parameters for the underlying type. For example, `tEnv::SharedArray<double> x(env, p);` creates an array of double-precision floating point values of length p on each processor, and directly registers it on the environment. The shared variables replace the `bsp_push_reg` and `bsp_pop_reg` primitives for registering and de-registering variables.

Put

As shown in Snippet 6.1, `PutValue(target, value, offset)` on an array puts a value into the array of the target processor, at an offset. Similarly, `Put(target, beginIterator, endIterator, offset)` puts all values from `beginIterator` up to (not including) `endIterator` into the array of the target processor, starting from a given offset. More variants of this operation will be added, but the basic idea is similar to the idea of BSPlib.

Get

Similar to `Put`, the `Get` primitive will also work on shared variables, with similar syntax.

Send

An example of the use of the `Send` primitive is shown in Snippet 6.2. The example can be used interchangeably with the inner product from Snippet 6.1. Instead of a shared variable, you now have a `SendQueue`, that contains structured messages. A queue can also consist of multiple types, for example `tEnv::SendQueue<size_t, double>` could be a queue containing an index and the value, but any number of types can be used in the queue. This replaces the `bsp_set_tagsize`, `bsp_send`, `bsp_qsize`, `bsp_get_tag` and `bsp_move` primitives from the BSPlib standard, as well as allowing more flexibility as multiple queues can coexist. The idea of structured queues was inspired by Bulk [BB17]; [BBB], as this corresponds more to `C++` data structures. As in the BSPlib standard, the order of the messages is not guaranteed, so if the computation requires the order of messages, you need to tag the messages, with for example the tag in the `size_t` of `tEnv::SendQueue<size_t, double>`.

Snippet 6.2. Inner product using `Send`

```
template<typename tEnv>
double InnerProductSend(tEnv &env,
                        std::vector<double> &x,
                        std::vector<double> &y)
{
```

```
    size_t p = env.Size();
    size_t s = env.Rank();

    // using SendQueue for partial inner products
    tEnv::SendQueue<double> partialInnerProducts(env);

    double alpha = 0.0;

    for (size_t i = 0, iEnd = x.size(); i < iEnd; ++i)
        alpha += x[i] * y[i];

    for (size_t t = 0; t < p; ++t)
        partialInnerProducts.Send(t, alpha);

    env.Sync();

    // Sum the partial inner products
    return std::accumulate(partialInnerProducts.begin(),
                           partialInnerProducts.end(),
                           0.0);
}
```

6.1.6 Sync

Finally, there is the familiar `Sync` primitive from BSPlib, with syntax `env.Sync()`. It synchronises the environment, meaning all the values that were scheduled for communication will be available in the environment once the `Sync` is done. Note that values communicated in the sub-environments will not be available after the synchronisation of the parent environment. Explicit synchronisation of the sub-environment is required for this. Synchronisation of an environment guarantees:

i) The receiving part of the `SendQueues` is considered to be processed and is cleared before receiving any other messages in it.

ii) All participants have written their buffer before it is communicated to other processors.

iii) All targets have received the payload scheduled with `Put` operations.

iv) All sources have retrieved the data requested by `Get` operations.

v) All participants have a combined buffer of `Send` that were sent to them by all participants, on a per-queue basis.

vi) All participants start the next superstep once the synchronisation is completed.

Barrier

An addition to the synchronisation primitives is the `env.Barrier()` primitive. It synchronises the processors without finalising the communication.

This is useful for benchmarking, as the `Barrier` primitive is much more lightweight, and still ensures all processors start at the same time with the benchmark.

The barrier is also used in the backend to synchronise threads using shared memory. The barrier algorithm is shown in Algorithm 6.1. It makes use of atomic[2] shared variables, to guarantee consistency.

Algorithm 6.1. Spinbarrier for synchronisation.

input : `superstepCounter` :an atomic shared variable indicating the current superstep,
`pLeft` :an atomic shared variable indicating the number of participants that have not yet reached the barrier.

procedure BARRIER
  `currentSuperstep` ← `superstepCounter`
  `afterDecrease` ← atomic decrease-and-fetch(`pLeft`)
  if `afterDecrease` equals 0 **then**
    reset `pLeft`
    atomic increase(`superstepCounter`)
  else
    spin while(`currentSuperstep` equals `superstepCounter`)
  end if
end procedure

6.1.7 Utilities

SyncLib will also be extended with utilities. At the moment there is a `Timer` utility, that can be used to record the time. It works similar to the `matlab` utility pair `tic` and `toc`: to reset the timer, you can call `timer.Tic()`, and to retrieve the elapsed time, you can call `timer.Toc()`. You can also choose the desired resolution of the timer: nano-, micro- or milliseconds, seconds, minutes, or hours.

6.1.8 Benchmarking and relabelling

Benchmarking and relabelling according to the optimal configuration will also be an integral part of SyncLib. In order to determine the optimal subset configuration, both the pairwise and environment-wide benchmarks described in Chapter 5 need to be part of the library. Several configurations can be passed to an optimisation function on the environment, to create d subset-environments for a d-dimensional cartesian distribution.

[2]Atomic variables are explained in Section 2.1.2.

6.2 Backend optimisation

There are several optimisations implemented in the backends, both in the MPI backend and the shared memory backend. These contribute to lower overall communication and synchronisation cost in any algorithm implemented using the SyncLib library.

6.2.1 Asynchronous MPI communication

The newer MPI standards support asynchronous communication primitives. The synchronisation of processors is done using asynchronous collective communication only. There are many buffers in the MPI backend that need to be communicated. By scheduling the communication asynchronously, we can perform the construction of other buffers in the meantime. For example, a `Get` request is buffered on the requesting side when the `Get` primitive is used. This buffer of requests (without payload data) is ready at the beginning of a synchronisation, so we can schedule it immediately. The same goes for the `Put` buffer containing addresses, offsets and the actual payload data. The asynchronous primitives return a `MPI_Request` handler, that can be used to wait for the request to complete. Instead of waiting *before* the put buffer communication is scheduled, we can wait *after* the other communication is scheduled. While the put buffer is communicated, the processors can construct the `Get` buffer with the actual payload data that needs to be retrieved. These asynchronous communications are used as much as possible, to fill the processor idle time as much as possible.

6.2.2 Maintaining the first-touch principle for shared memory buffers

Another important concept is the first-touch principle [Ter+08]: the processor that touches the memory first (writes a value to it), will be the owner of that memory. This means that this part of the memory will reside in the fast cache layers of memory for that processor. Other processors trying to read the memory will have to go through slower[3] shared memory to obtain the value. By separating the buffer for processors more distinctly, and letting the processor that owns a receive buffer initialise and fill it with zeros, memory ownership will become more clear to the CPU. In single-socket architectures, ownership can more easily be (implicitly) transferred to the processor currently using the memory, but this optimisation will certainly not harm performance in that scenario.

To better enforce this ownership of buffers in the shared memory setting, every processor gets a duplicate of the environment, where only the barrier is shared, but send and receive buffers are separate for each processor, and each processor has a reference to the environments of the

[3] Slower compared to the fast cache memory

other processors. Every processor is only allowed to write to buffers that reside in its own duplicate of the environment, and allowed to read from the environment buffers from other processors during synchronisation.

6.2.3 Using the Ping-Pong partner function for shared memory communication

Reading shared memory that is owned by a different processor causes the cache coherency mechanism of the computer to synchronise the shared memory. The cache coherency mechanism forces the processor owning the data to flush it to shared memory, and the processor trying to read the data to retrieve it from shared memory after the flush. This causes much more latency for every read operation. While we cannot control this behaviour directly, we can try to reduce the number of times it happens during synchronisation, by working pairwise as much as possible. This is done in SyncLib by using the simple version of the Ping-Pong partner function from Equation (5.6). In the worst case, it does not help the cache coherency mechanism, but it only costs a few more flops for the synchronisation, namely $O(p)$, the number of processors that is used. In the best case, the cache coherency mechanism can work pairwise, instead of over the whole set of processors, significantly reducing latency.

6.3 Future extensions

6.3.1 Higher-level communication

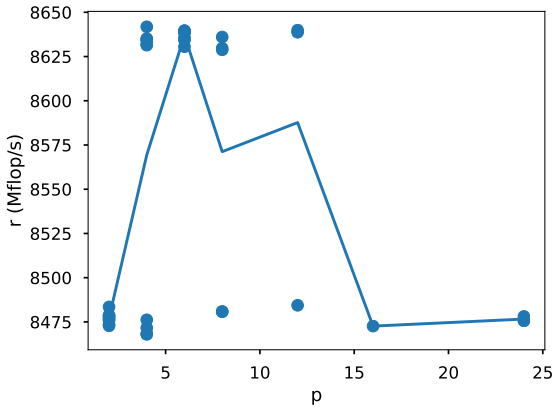
Something that is not yet a part of SyncLib, but will be implemented after this thesis, is higher-level communication. An example of higher-level communication would be broadcasting a value or array of values into a shared value. Higher-level communication will relieve the program text from managing targets, sizes and offsets in collective communication. Other examples would be scatter (divide an array of values over all processors in an environment) and gather (combining an array of values that is divided over processors into a local array).

6.3.2 Parallel reduce

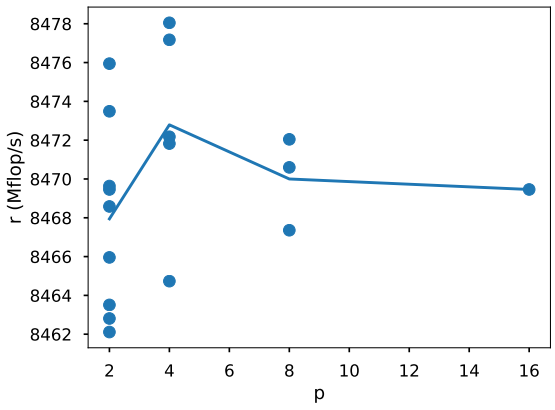
Another high-level algorithmic construct would be parallel reduce operations: reducing input variables that are divided over processors to a single value. This will also be implemented after this thesis, but can already be used in writing algorithms

6.3.3 Visualisations

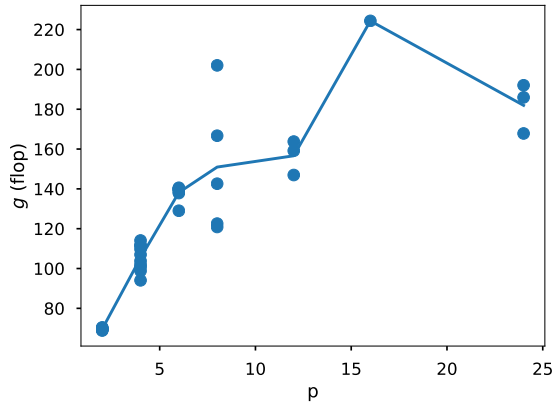
Similar to Zefiros-BSPLib, the SyncLib library will be providing visualisation tools to visualise communication volumes and time per superstep. At the moment, this is not built into the library, but it will be added later.



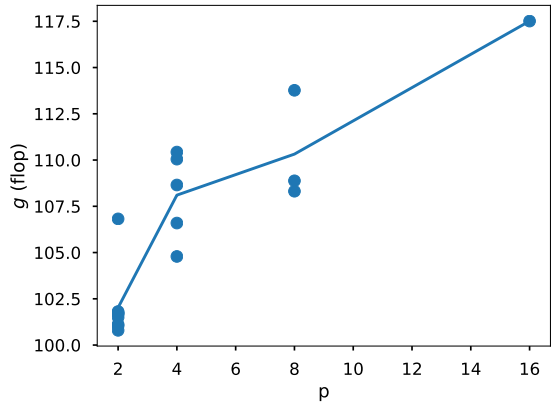
(a) Measurement of r on a single node with p processors.



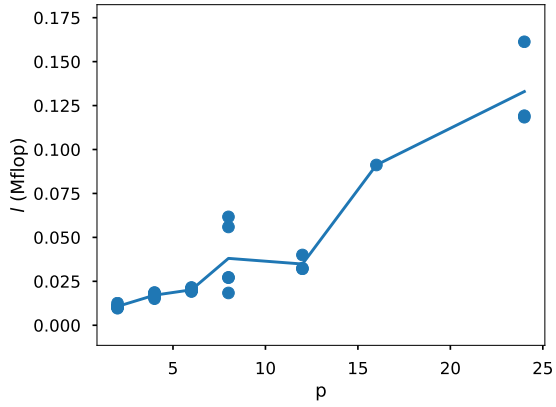
(b) Measurement of r with p nodes, each with only one processor.



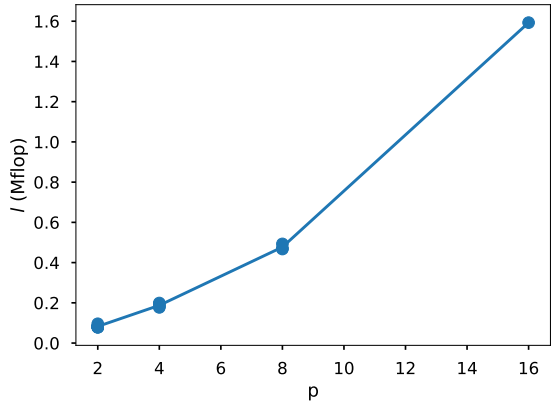
(c) Measurement of g on a single node with p processors.



(d) Measurement of g with p nodes, each with only one processor.



(e) Measurement of l on a single node with p processors.



(f) Measurement of l with p nodes, each with only one processor.

Figure 6.3: Measurement of several values of r, g, l for different p, using SyncLib. Both g, l are normalised from the measured r to $r_0 = 8528.27$, the average r over all runs.

6.4 Speed comparison

Generalisation in a software library usually introduces overhead in the cost. As SyncLib is generalised to hierarchical machines, it is interesting to see how it compares to for example MulticoreBSP version 1.0.2 [Yze14]. We performed a similar benchmark to Section 5.1 with both libraries on Cartesius, the Dutch national supercomputer, which was shown in Figure 5.2. As SyncLib has a different syntax, the benchmark was ported to the new syntax. The resulting measurements of h-relations followed the same linear relation as Figure 5.1. The r, g, l values corresponding to these measurements are shown in Figure 6.3. We still see the similar fluctuations in r, but the average value of r seems to be significantly lower, only 2/3 of the value that was measured in the BSPedupack [Bis04] benchmark of the same machine. This is not what would be expected, as the measurement of r is without interference of communication and synchronisation. This can however be explained by compiler optimisations: MulticoreBSP is compiled with an older `C` compiler, while SyncLib is compiled with one of the newer `C++` compilers. Data structures and operations performed on the data are optimised differently by the compilers, so in order to get the same performance, we have to choose a different data structure. For now, the results are made comparable by normalising the g, l measurement with the same $r_0 = 8528.27$. With knowledge of the fact that a single node consists of two sockets, we can spot two patterns of growth in both g, l for a single node: g starts low (very low compared to MulticoreBSP) and grows steeply within the first socket. Then for $p = 16$, only 4 processors of the second socket are used, and we can see that this imbalance spikes the communication cost. As we predicted, the worst-case communication cost will dominate the overall communication. When we use all processors of the second socket, the average cost is compensated and reduces again. Still, the overall g is lower than that of MulticoreBSP. Similar, but inverted growth patterns can be seen in the measurement of l on a single node: the growth is moderate within the first socket, is still spiked slightly for $p = 16$, but grows at a larger rate as we use more processors in the second socket.

If we look at the case where we use a single processor of each node, the measurement is much more stable. The spikes in r are less prominent, g grows linearly (up to some artefacts), and the growth in l is stable, but again superlinear, which is all the more clear due to the stable measurement. What is perhaps the most interesting to see, is that g is very low, comparable to the first socket of the single node measurement, but much lower than when we started using the second socket. What may come to mind is that we could just flatten the hierarchy to profit from this lower cost when we use the multiple sockets. What has to be noted however, is that the growth in g is still almost linear. In case we use a large number of nodes, it becomes very beneficial to use intra-node communication between processors using shared-memory, in order to keep

the inter-node communication cost, as well as the synchronisation cost, as low as possible. Moreover, l is bigger by almost a factor 10 for the inter-node synchronisation, so the hierarchical approach will still very much reduce the synchronisation time. If we compare the distributed version to MulticoreBSP, it is interesting to see that while the communication volume is roughly the same (up to some packing of the messages), the communication cost is halved. This is probably due to the asynchronous MPI primitives that were used in SyncLib.

Finally, the value of l for distributed nodes is much larger than the one we measured for MulticoreBSP, thus there is still room for improvement. At the moment, there are many collective communications introducing implicit intermediate synchronisations for exchanging the sizes of buffers and exchanging the actual buffers. The sizes are not yet combined into a single payload, so this certainly leaves room for improvement in the future versions of SyncLib.

7 Hierarchical Parallel Algorithms

In order to implement a parallel algorithm in a hierarchical manner, we need to identify the most time-consuming parts of the algorithm, and parametrise the sub-environment(s) on that part of the algorithm. We can do this by looking at the dominating term in the cost expression. By parametrising the sub-environment, we can always fall back to the (local) sequential algorithm for that part, in case there is no parallel sub-environment. This does not mean that part of the algorithm will be executed fully sequentially, just that the node will not run that part in a sub-environment.

7.1 Hierarchical matrix-matrix multiplication

We are now able to transform the matrix-matrix multiplication algorithm from Section 3.1 to a hierarchical algorithm. The computationally most expensive part clearly is the multiplication of sub-matrices in superstep (1) of Algorithm 3.1. Algorithm 7.1 shows the transformed algorithm. The Reduce_+ operation was explained in Section 3.1. The main changes are visible in the input and in superstep (1); we have a list of sub-environments in which we want to run the algorithm, and we invoke a sub-algorithm in superstep (1), that still needs to be defined for (sensible) sub-environments. The algorithms for the sub-environments are given in Algorithm 7.4 and Algorithm 7.3. Note that the algorithm for the `SharedMemoryBSP` environment is recursively applicable, thus we can recursively run the algorithm in sub-environments, but while smaller sub-environments have smaller communication cost, this generally does not give us any benefits, as we are introducing more synchronisations.

Where the algorithm can still benefit from the new model, is in splitting the (sub-) environment, as described in Section 4.2.3. Note that in superstep (0) of Algorithm 7.1, we have communication between nodes that either share the same (s, u), or the same (u, t). In superstep (2), there is communication between nodes that share the same (s, t). We can now make three splits to facilitate this communication pattern: q subset-environments E_{su} that share the same (s, u), in which nodes are identified by $P_{su}(t) = P(s, t, u)$, and similarly q subset-environments E_{ut} that share

the same (u,t), with $P_{ut}(s) = P(s,t,u)$, and q subset-environments E_{st} that share the same (s,t), with $P_{st}(u) = P(s,t,u)$. A summary of the transformation is shown in Algorithm 7.2. The implemented algorithm is available in the SyncLib repository[1].

Algorithm 7.1 Hierarchical three-dimensional matrix multiplication.

input : $\mathbf{A} : n \times m$ matrix, distr $(\mathbf{A}) = \phi_A$,
$\mathbf{B} : m \times k$ matrix, distr $(\mathbf{B}) = \phi_B$,
$[E_1, E...]$: a list of sub-environment types,
output : $\mathbf{C} : n \times k$ matrix, $\mathbf{C} = \mathbf{AB}$, distr $(\mathbf{C}) = \phi_C$.
with $p_0 = q_0^3, q_0 \in \mathbb{N}$,
(refer to Definition 1 for distributions)

{Replicate $\mathbf{A}_{su}$ over $P(s,*,u)$} ▷ superstep (0)
for all $(i,j) \in N \times M \ \wedge \ \phi_A(i,j) = (s,t,u)$ **do**
put $\mathbf{A}_{i,j}$ in $P(s,*,u)$

{Replicate $\mathbf{B}_{ut}$ over $P(*,t,u)$}
for all $(i,j) \in M \times K \ \wedge \ \phi_B(i,j) = (s,t,u)$ **do**
put $\mathbf{B}_{i,j}$ in $P(*,t,u)$

{Compute the contribution to $\mathbf{C}$ of $P(s,t,u)$} ▷ superstep (1)
$\widetilde{\mathbf{C}}_{stu} \leftarrow$ `MatrixMultiply<`$E_1, E...$`>`$(\mathbf{A}_{su}, \mathbf{B}_{ut})$

{Sum the contributions to $\mathbf{C}$ over $P(s,t,*)$} ▷ superstep (2)
for all $(i,j) \in N \times K \ \wedge \ \phi_C(i,j) = (s,t,u)$ **do**
$C(i,j) = \text{Reduce}_+(\widetilde{\mathbf{C}}(i \mod b_n, j \mod b_k), P(s,t,*))$

Algorithm 7.2 Subset-environment version of Algorithm 7.1.

This algorithm is exactly the same as Algorithm 7.1, except we replace $P(s,*,u)$ by $P_{su}(*)$, $P(*,t,u)$ by $P_{ut}(*)$ and $P(s,t,*)$ by $P_{st}(*)$. This also means that superstep (0) is split into two smaller supersteps, operating in two different subset environments, namely $P_{su}(*)$ and $P_{ut}(*)$, and superstep (2) now works in subset environment $P_{st}(*)$.

The sub-algorithm `MatrixMultiply` for different sub-environments is shown in Algorithm 7.3 and Algorithm 7.4. Note that while the environments for the sub-algorithms are explicitly stated, the main environment can still be either based on shared memory, or on distributed memory. While Algorithm 7.4 has no communication in the E_{su} and E_{ut} subset-environments, we can still update Algorithm 7.4 with a split environment for the final superstep, as shown in Algorithm 7.5.

[1]Located at https://github.com/Zefiros-Software/SyncLib/tree/master/matmat.

Algorithm 7.3 `MatrixMultiply` for $E_1 = \texttt{NoBSP}$.

input : $\mathbf{A} : n_1 \times m_1$ matrix,
$\mathbf{B} : m_1 \times k_1$ matrix,
output : $\mathbf{C} : n_1 \times k_1$ matrix.

{Compute $\mathbf{C}$ using any (optimised) sequential algorithm}
$\mathbf{C} = \mathbf{AB}$

Algorithm 7.4 `MatrixMultiply` for $E_1 = \texttt{SharedMemoryBSP}$.

input : $\mathbf{A} : n_1 \times m_1$ matrix in shared memory,
$\mathbf{B} : m_1 \times k_1$ matrix in shared memory,
$[E_2, E...]$: a list of sub-environment types to run the algorithm in,
output : $\mathbf{C} : n_1 \times k_1$ matrix in shared memory.
with $p_q = q_1^3, q_1 \in \mathbb{N}$,
(refer to Definition 1 for distributions).

{Make subviews of matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$} ▷ superstep (0)
$\mathbf{A}_{su} \leftarrow \mathbf{A}(s \cdot b_n : (s+1) \cdot b_n - 1, \quad u \cdot b_m : (u+1) \cdot b_m - 1)$
$\mathbf{B}_{ut} \leftarrow \mathbf{B}(u \cdot b_m : (u+1) \cdot b_m - 1, \quad t \cdot b_k : (t+1) \cdot b_k - 1)$

{Compute the contribution to $\mathbf{C}$ of $P(s,t,u)$}
$\widetilde{\mathbf{C}}_{stu} \leftarrow \texttt{MatrixMultiply<}E_2, E...\texttt{>}(\mathbf{A}_{su}, \mathbf{B}_{ut})$

{Sum the contributions to $\mathbf{C}$ over $P(s,t,*)$} ▷ superstep (1)
for all $(i,j) \in N \times K \ \wedge \ \phi_C(i,j) = (s,t,u)$ **do**
$C(i,j) = \text{Reduce}_+(\widetilde{\mathbf{C}}(i \mod b_n, j \mod b_k), P(s,t,*))$

Algorithm 7.5 Subset-environment version of Algorithm 7.4.

This algorithm is exactly the same as Algorithm 7.4, except we replace $P(s,t,*)$ by $P_{st}(*)$. This also means that superstep (2) now works in subset environment $P_{st}(*)$.

Now that we have both the split and unsplit version of the algorithm and the corresponding algorithms for sub-environments, we can analyse the cost in terms of the extended BSP cost. As the algorithm has a sub-algorithm, the cost will have a sub-expression for the cost of this

sub-algorithm.

$$T_{\texttt{MatMat7.1}} = g_0 \cdot \left(\frac{nm}{q_0^3} + \frac{mk}{q_0^3}\right) \cdot (q_0 - 1) + l_0 \qquad \text{(superstep (0))} \quad (7.1)$$

$$+ T_{\texttt{MatMatSub}}\texttt{<}E_1, E...\texttt{>} + l_0 \qquad \text{(superstep (1))} \quad (7.2)$$

$$+ g_0 \cdot \frac{nk}{q_0^3} \cdot (q_0 - 1) + l_0 \qquad \text{(superstep (2))} \quad (7.3)$$

$$+ \frac{nk}{q_0^3} \cdot (q_0 - 1) + l_0 \qquad \text{(superstep (2*))} \quad (7.4)$$

This cost is easy to analyse: in superstep (0) we need to broadcast our part of $\mathbf{A}$ of size $\frac{nm}{q_0^3}$ and our part of $\mathbf{B}$ of size $\frac{mk}{q_0^3}$ to all members that either share the same (s, u) or (u, t) label. In both cases, there are $(q_0 - 1)$ such others. Then every node runs the sub-algorithm in the given sub-environment in superstep (1). Superstep (2) actually consists of a communication phase and computation phase. In the analysis, superstep (2) is just the communication: we need to send our contribution to $\mathbf{C}$ to the owner of the corresponding elements of $\mathbf{C}$. The participants that share the (s, t) label each have a contribution to each other's elements of $\mathbf{C}$, and each of them owns $\frac{nk}{q_0^3}$ elements of $\mathbf{C}$. Thus, each node needs to communicate $q_0 - 1$ parts of size $\frac{nk}{q_0^3}$. Finally, each node needs to sum the $q_0 - 1$ parts that it received with the part it already has, in superstep (2*). We can simplify the total cost with an upperbound by noting $q_0 - 1 < q_0$, so

$$T_{\texttt{MatMat7.1}} < 4l_0 + g_0 \frac{nm + mk + nk}{q_0^2} + T_{\texttt{MatMatSub}}\texttt{<}E_1, E...\texttt{>} + \frac{nk}{q_0^2}. \quad (7.5)$$

As the split subset-environment algorithm is pretty similar, the translation of the cost is trivial. We only need to take care that we put the correct synchronisation cost parameter on computation supersteps. As a rule of thumb, we can use the average of the synchronisation cost parameter of the preceding and succeeding communication supersteps as the synchronisation cost of a computation superstep, with the exception of the first and final superstep. In case the first superstep is a computation superstep, we use the synchronisation cost parameter of the succeeding superstep, and analogously we use the cost of the preceding superstep in case the final superstep is a computation superstep. The cost then becomes

$$T_{\texttt{MatMat7.2}} < l_0^{(su)} + l_0^{(ut)} + g_0^{(su)} \frac{nm}{q_0^2} + g_0^{(ut)} \frac{mk}{q_0^2} \quad (7.6)$$

$$+ T_{\texttt{MatMatSub}}\texttt{<}E_1, E...\texttt{>} + \frac{1}{2}\left(l_0^{(ut)} + l_0^{(st)}\right) \quad (7.7)$$

$$+ g_0^{(st)} \frac{nk}{q_0^2} + l_0^{(st)} \quad (7.8)$$

$$+ \frac{nk}{q_0^2} + l_0^{(st)}. \quad (7.9)$$

We can further simplify this cost again by defining

$$g_0^* := \max\left(g_0^{(su)}, g_0^{(ut)}, g_0^{(st)}\right), \tag{7.10}$$

$$l_0^* := \max\left(l_0^{(su)}, l_0^{(ut)}, l_0^{(st)}\right). \tag{7.11}$$

Then $g_0^{(su)}, g_0^{(ut)}, g_0^{(st)} \leq g_0^*$, and $l_0^{(su)}, l_0^{(ut)}, l_0^{(st)} \leq l_0^*$. In general, we have $g_0^* \leq g_0$, and in case of a superlinear relation of the synchronisation cost with the number of processors (which is quite common), we should at least have $l_0^* \leq l_0/q_0^2$. Thus, we can simplify the cost to

$$\begin{aligned} T_{\texttt{MatMat7.2}} &< 5l_0^* + g_0^* \frac{nm + mk + nk}{q_0^2} + T_{\texttt{MatMatSub}}\texttt{<}E_1, E...\texttt{>} + \frac{nk}{q_0^2} \\ &\leq \frac{5l_0}{q_0^2} + g_0 \frac{nm + mk + nk}{q_0^2} + T_{\texttt{MatMatSub}}\texttt{<}E_1, E...\texttt{>} + \frac{nk}{q_0^2}. \end{aligned} \tag{7.12}$$

What remains is to analyse the cost for the sub-algorithms. For Algorithm 7.3, the cost is simply

$$T_{\texttt{MatMatSub7.3}}\texttt{<NoBSP}, E...\texttt{>} = 2n_1m_1k_1, \tag{7.13}$$

which is the cost of sequential matrix multiplication. We can also apply sequential algorithms like Strassen matrix multiplication to further reduce the cost. This is easy for $m_1 = k_1 = n_1$; then the asymptotic cost becomes $O(n_1^{2.807})$. This is also achievable (with some effort) for sizes that are not equal. The remaining sub-algorithms are the unsplit and split version for `SharedMemoryBSP`. The cost for the unsplit version is

$$T_{\texttt{MatMatSub7.4}}\texttt{<SharedMemoryBSP}, E_2, E...\texttt{>} \tag{7.14}$$

$$= T_{\texttt{MatMatSub}}\texttt{<}E_2, E...\texttt{>} + l_1 \qquad \text{(superstep (0))} \tag{7.15}$$

$$+ g_1 \cdot \frac{n_1k_1}{q_1^3}(q_1 - 1)) + l_1 \qquad \text{(superstep (2))} \tag{7.16}$$

$$+ \frac{n_1k_1}{q_1^3}(q_1 - 1) + l_1, \qquad \text{(superstep (2*))} \tag{7.17}$$

and the split version has cost

$$T_{\texttt{MatMatSub7.5}}\texttt{<SharedMemoryBSP}, E_2, E...\texttt{>} \tag{7.18}$$

$$= T_{\texttt{MatMatSub}}\texttt{<}E_2, E...\texttt{>} + l_1^{(st)} \qquad \text{(superstep (0))} \tag{7.19}$$

$$+ g_1^{(st)} \cdot \frac{n_1k_1}{q_1^3}(q_1 - 1)) + l_1^{(st)} \qquad \text{(superstep (2))} \tag{7.20}$$

$$+ \frac{n_1k_1}{q_1^3}(q_1 - 1) + l_1^{(st)}. \qquad \text{(superstep (2*))} \tag{7.21}$$

We can now combine these cost expressions depending on the sub-environments we have. Suppose we implement all algorithms in the subset-split version, and execute them on the environment configuration $[E_0, E_1 = \texttt{SharedMemoryBSP}, E_2 = \texttt{NoBSP}]$, where E_0 is either the

distributed-memory or shared-memory BSP implementation. Recall that as we descend into sub-algorithms, we continue to work with the size that was allocated to the processor in the parent environment, thus we have $n_1 = \frac{n}{q_0}, m_1 = \frac{m}{q_0}, k_1 = \frac{k}{q_0}$, and $n_2 = \frac{n_1}{q_1}, m_2 = \frac{m_1}{q_1}, k_2 = \frac{k_1}{q_1}$. The number of processors in the main environment was $p_0 = q_0^3$, and in each sub-environment $p_1 = q_1^3$. Then the combined cost becomes

$$\begin{aligned} T_{\texttt{MatMat7.2}} &\leq \frac{5l_0}{q_0^2} + g_0\frac{nm+mk+nk}{q_0^2} + \frac{nk}{q_0^2} \\ &\quad + \left(3l_1^{(st)} + (g_1^{(st)}+1)\cdot\frac{n_1k_1}{q_1^2} + \left(2n_2m_2k_2\right)\right) \\ &= \frac{5l_0}{q_0^2} + 3l_1^{(st)} + g_0\frac{nm+mk+nk}{q_0^2} + (g_1^{(st)}+1)\cdot\frac{nk}{(q_0q_1)^2} + 2\frac{nmk}{(q_0q_1)^3} \\ &= \frac{5l_0}{p_0^{2/3}} + 3l_1^{(st)} + g_0\frac{nm+mk+nk}{p_0^{2/3}} + (g_1^{(st)}+1)\cdot\frac{nk}{(p_0p_1)^{2/3}} + 2\frac{nmk}{p_0p_1}. \end{aligned}$$

The bulk of the computational cost is divided by the product of the number of nodes, and the number of processors internally in the node, as expected.

Suppose we were to look at all processors at the same time, so one distributed-memory environment of size p_0p_1. At first glance, it seems that we only have $\widetilde{g}\cdot\frac{nm+mk+nk}{(p_0p_1)^2}$ as the communication cost. However, the processors in a node share the same network connection to other nodes, so the actual communication volume between *nodes* is still the same, which would probably reflect in the communication cost parameter $\widetilde{g}$. Moreover, the number of participants partaking in the global communication is p_0p_1, so this may worsen the latency (super)linearly by a factor p_1.

7.2 Hierarchical LU decomposition

Another common algorithm that has many applications, is the LU decomposition algorithm. It decomposes a square $n \times n$ matrix $\mathbf{A}$ into an upper-triangular and lower-triangular matrix, such that $\mathbf{A} = \mathbf{LU}$, where $\text{diag}(\mathbf{L}) = 1$. Moreover, due to the triangular structure and the diagonal of $\mathbf{L}$ being equal to 1, we can store them together in one matrix, by leaving out the diagonal of $\mathbf{L}$ and implicitly assuming that it is 1.

The basic LU decomposition algorithm is an extension of Gaussian elimination (without row swaps): not only do we store the rows in $\mathbf{U}$ after subtracting a linear multiple of a row, but we also store the factors we subtracted in a column of $\mathbf{L}$. The algorithm is pretty straightforward, as shown in Algorithm 7.6.

An important application of the LU decomposition is linear solvers. Suppose we want to solve $\mathbf{A}\vec{x} = \vec{b}$. This corresponds to solving $\mathbf{L}\vec{y} = \vec{b}$, and then solving $\mathbf{U}\vec{x} = \vec{y}$, since then it holds that $\mathbf{LU}\vec{x} = \mathbf{L}\vec{y} = \vec{b}$. Solving these two systems is easy due to the triangular structure. We solve $\mathbf{L}\vec{y} = \vec{b}$ by forward substitution: we process the rows in increasing

Algorithm 7.6. Sequential LU decomposition without pivoting.

input : $\mathbf{A} : n \times n$ matrix,
output : $\mathbf{A} : n \times n$ matrix, $\mathbf{A} = (\mathbf{L} - \mathbf{I}) + \mathbf{U}$,
$\mathbf{L} : n \times n$ lower triangular matrix,
$\mathbf{U} : n \times n$ upper triangular matrix,
where $K(k) = \{k+1, k+2, ..., n-1\}$.

for $0 \leq k < n$ **do**
  for all $i \in K(k)$ **do**
    $a_{ik} := a_{ik}/a_{kk}$

  for all $(i, j) \in K(k) \times K(k)$ **do**
    $a_{ij} := a_{ij} - a_{ik}a_{kj}$

order, leaving only one unknown per row, and solving a linear equation with one unknown is trivial. Analogously, we solve $\mathbf{U}\vec{x} = \vec{y}$ by solving the rows in decreasing order. At first sight, this does not seem simpler than performing Gaussian elimination to solve the linear system, as we will be performing a very similar algorithm during the decomposition, but once we need to solve for multiple values of $\vec{b}$, this approach becomes much more attractive.

The variant that will be discussed in this section is the LU decomposition with partial pivoting. At stage k of the LU decomposition with partial pivoting, we find the largest (in absolute value) element a_{rk}, and swap rows r and k, thus performing row permutations on the matrix. Instead of $\mathbf{A} = \mathbf{LU}$, we now have $\mathbf{PA} = \mathbf{LU}$, where $\mathbf{P}$ is a permutation matrix. Partial pivoting prevents many of the cases where the basic LU decomposition is halted because of division-by-zero, and is also numerically more stable. Without going into detail, we can already notice that without pivoting, we may be dividing by very small numbers, and small errors are amplified. To solve $\mathbf{A}\vec{x} = \vec{b}$ in this new system, we simply solve $\mathbf{L}\vec{y} = \mathbf{P}\vec{b}$ and $\mathbf{U}\vec{x} = \vec{y}$. The LU decomposition with partial pivoting is also used in benchmarking the top 500 supercomputers of the world, using the High-Performance LINPACK (HPL) benchmark [JPA03].

The parallelisation of the LU decomposition is extensively discussed in [Bis04]. There is a lot of similarity to the high-performance variant that is written out in Algorithm 7.7, so this algorithm will be referred to for clarification of the parallelisation. The main difference is that the high performance version works blockwise: a small amount of work is done first for every block, and the bulk of the work is postponed. Then the remaining work is done after the block is finished. This causes the double loop in the high-performance algorithm.

Algorithm 7.7 High-performance parallel LU decomposition.

input : $\mathbf{A} : n \times n$ matrix, $\text{distr}(\mathbf{A}) = M \times N$ cyclic, $\mathbf{A} = \mathbf{A}^{(0)}$
$[E_1, E...]$: a list of sub-environment types,
output : $\mathbf{A} : n \times n$ matrix, $\mathbf{A} = (\mathbf{L} - \mathbf{I}) + \mathbf{U}$, $\text{distr}(\mathbf{A}) = M \times N$ cyclic,
$\mathbf{L} : n \times n$ lower triangular matrix,
$\mathbf{U} : n \times n$ upper triangular matrix,
$\vec{\pi}$: permutation vector of length n, redundantly stored on every node, such that $a^{(0)}_{\vec{\pi}(i),j} = (\mathbf{LU})_{ij}$ for $0 \leq i, j < n$,
with $M, N, b \in \mathbb{N}, p = MN, K_0 = \{0, b, 2b, ..., \left\lfloor \frac{n-1}{b} \right\rfloor \cdot b\}$
$K_1(k_0, k_1) = \{k_0, k_0 + 1, ..., k_1 - 1\}$,
$K(k) = \{k, ..., n-1\}, k_1 = \min(k_0 + b, n)$
$\vec{a}_i(k, k_1) = \{a_{ij} : k \leq j < k_1 \wedge j \mod N = t\}$,
$\vec{b}_j(k, k_1) = \{a_{ij} : k \leq i < k_1 \wedge i \mod M = s\}$,
$(\mathbf{A}^{(k)}_{11})_{ij} = a_{ij}$, for all $k + 1 \leq i, j < k_1$
$\wedge\, i \mod M = s \wedge j \mod N = t$.

for all $i \in K$ **do**
  $\vec{\pi}_i := i$

for $k_0 := 0$ to $\left\lfloor \frac{n-1}{b} \right\rfloor \cdot b\}$ step b **do**
  initialise $\mathbf{L}_{11} : b \times b$ to 0
  initialise $\mathbf{U}_{12} : b \times \left\lceil \frac{n-k_1}{N} \right\rceil$ to 0
  initialise $\mathbf{L}_{21} : \left\lceil \frac{n-k_1}{M} \right\rceil \times b$ to 0
  initialise Δ to an empty list of pairs
  {Perform b stages in only columns $k_0 \leq k < k_1$}
  for $k_0 \leq k < k_1$ **do**
    if $k \mod N = t$ **then**
      {Locally determine pivot element} ▷ superstep (0)
      $r_s := \arg\max(|a_{ik}| : k \leq i < n \wedge i \mod M = s))$
      {Share with the node column} ▷ superstep (1)
      put r_s, and $a_{r_s,k}$ as α_s in $P_t(*)$
      {Determine global pivot element} ▷ superstep (2)
      $s_{\max} := \arg\max(|\alpha_q| : 0 \leq q < M)$
      $r := r_{s_{\max}}$
      {Divide column k in $\mathbf{A}_{11}, \mathbf{A}_{21}$}
      for all $i \in K(k) \wedge i \mod M = s \wedge i \neq r$ **do**
        $a_{ik} := a_{ik} / \alpha_{s_{\max}}$

      {Share r with the node row} ▷ superstep (3)
      put r in $P_s(*)$

end if
{Swap the row within $\mathbf{A}_{11}, \mathbf{A}_{21}$} ▷ superstep (4)
if $k \mod M = s$ **then**
send $(k, \vec{a}_k(k_0, k_1))$ to $P_t(k \mod M)$ into S_{swap}
if $r \mod M = s$ **then**
send $(r, \vec{a}_r(k_0, k_1))$ to $P_t(r \mod M)$ into S_{swap}
{Receive the row and store it} ▷ superstep (5)
for all $(i, \vec{a}) \in S_{\text{swap}}$ **do**
$\vec{a}_i(k, k_1) := \vec{a}$

swap $(\vec{\pi}_k, \vec{\pi}_r)$, append (k, r) to Δ
{Broadcast row and column k} ▷ superstep (6)
broadcast $\vec{a}_k(k_0, k_1)$ to $P_t(*)$
broadcast $\vec{b}_k(k+1, n)$ to $P_s(*)$
{Update the remaining block of $\mathbf{A}_{11}$} ▷ superstep (0')
$\mathbf{A}_{11}^{(k)} := \mathbf{A}_{11}^{(k)} - \vec{b}_k(k+1, k_1)^T \cdot \vec{a}_k(k+1, k_1)$
$\mathbf{A}_{21}^{(k)} := \mathbf{A}_{21}^{(k)} - \vec{b}_k(k_1, n)^T \cdot \vec{a}_k(k+1, k_1)$
{Update our temporary copies}
store $\vec{a}_k(k_0, k)$ in $\mathbf{L}_{11}$
store $\vec{b}_k(k_1, n)$ in $\mathbf{L}_{21}$
... to be continued

The implementation of the algorithm can be found in the SyncLib repository[2]. The parallel LU decomposition uses the square cyclic distribution, that is, we have $p = M \cdot N$ processors that are labelled with two-dimensional labels (s, t), where $0 \leq s < M$ and $0 \leq t < N$. Element a_{ij} then belongs to $\phi(i, j) = (\phi_0(i), \phi_1(j)) = (i \mod M, j \mod N)$. Problems that need to be solved due to the distributed data are: determining the pivot, swapping the pivot row r with row k at stage k, division by the pivot element, and updating the elements a_{ij} with $k < i, j < n$. Determining the pivot at stage k is done by cooperation of processors in processor column $k \mod N$: each processor locally determines a pivot candidate, and broadcasts the index along with the value to all processors in the processor column. They can then redundantly determine the global pivot element, and divide the column by this element. In Algorithm 7.7, this corresponds to supersteps (0) and (1), and these are exactly the same in both the high-performance and the normal parallel algorithm. Then processors owning row k (so every processor in processor row $k \mod M$) swap their elements with processors in the same processor column, owning row r, by putting the elements in the memory of the other processor. This corresponds to supersteps (4) and (5). However, in the normal parallel algorithm the entire row is swapped at this point, including the part in $\mathbf{A}_{10}$ and $\mathbf{A}_{12}$, see Figure 7.1. To update the remaining part of the matrix,

[2]Located at https://github.com/Zefiros-Software/SyncLib/tree/master/lu.

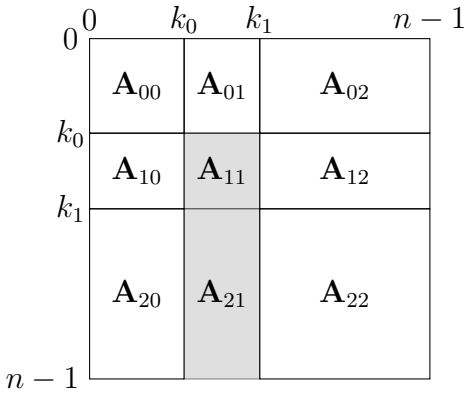


Figure 7.1: Visualisation of the submatrices involved in the LU decomposition with block size b.

the processor that owns elements a_{ij} with $k < i, j < n$ needs a_{ik}, a_{kj} to update the element. Thus, the elements in column k need to be broadcast to all processors in the same processor row, and the elements in row k need to be broadcast to processors in the same processor column. This corresponds to superstep (6), Again, the row is only partially communicated for the high performance algorithm, whereas it should be completely communicated in the normal parallel algorithm. The column is communicated completely in both algorithms. It may be worthwhile to use a *two-phase broadcast*, to overcome some of the communication imbalance in this broadcast. A two-phase broadcast first scatters the elements to all participants of the broadcast, and then all participants broadcast the received elements to every other participant. Then every processor can update its part of the remaining matrix, as shown in superstep (0'). Note that this superstep does not require a synchronisation, and is therefore merged with (0) due to the loop iteration.

In a birds-eye view, these are all the ingredients for a basic parallel LU decomposition with partial pivoting. As suggested in [Bis04] in an exercise, we can take performance a step further by processing the matrix in blocks of size b. Not all flops are equal. If we leverage optimised linear algebra routines (BLAS), the computation will be much more optimised than handwritten loops, not only in terms of loop iteration logic, but also in terms of cache optimality. This is the idea behind this high-performance version of the LU decomposition. The rest of the algorithm is continued on **page 58**. We first update the columns $k_0 \leq k < k_1 = k_0 + b$, where k_0 is the current multiple of b, and postpone row swaps in columns $0 \leq j < k_0$ and columns $k_1 \leq j < n$, as well as the matrix update of $\mathbf{A}(k_0 : n-1, k_1 : n-1)$. We can then leverage matrix multiplication to compute $a_{ij} := a_{ij} - \sum_{k=k_0}^{k_1-1} a_{ik}a_{kj}$ for $k_1 \leq i, j < n$. The sub-matrices involved are shown in Figure 7.1. The marked sub-matrices $\mathbf{A}_{11}, \mathbf{A}_{21}$ are updated for the stages $k_0 \leq k < k_1$. With the extended model, we can leverage sub-environments and split subset-environments in order to

reduce communication and synchronisation cost. Due to the $M \times N$ cyclic distribution, we have only communication in either a row of nodes, or column of nodes. We can use the optimised subset-environments explained in Section 5.2.2 to further reduce communication and/or synchronisation cost.

The b swaps that are performed during these stages can then be combined into a permutation. Instead of performing the swaps in sequence, we can then send the rows according to the permutation. As an example, suppose we have an index sequence $[i_0, i_1, i_2, i_3, i_4, i_5]$, and performs swaps of element pairs $(0, 4), (1, 3), (2, 4), (3, 5)$. The index sequence then undergoes the following stages:

$$[i_0, i_1, i_2, i_3, i_4, i_5] \overset{(0,4)}{\rightarrow} [i_4, i_1, i_2, i_3, i_0, i_5] \overset{(1,3)}{\rightarrow} [i_4, i_3, i_2, i_1, i_0, i_5]$$
$$\overset{(2,4)}{\rightarrow} [i_4, i_3, i_0, i_1, i_2, i_5] \overset{(3,5)}{\rightarrow} [i_4, i_3, i_0, i_5, i_2, i_1]$$

The owners of the row with index i_4 can just send row i_4 to the original owners (in the same processor column) of i_0, the owners of row i_3 to the original owners of row i_1 etc. This is then done for matrices $\mathbf{A}_{10}$ and $\mathbf{A}_{12}$, and all rows in $\mathbf{A}_{20}$ and $\mathbf{A}_{22}$ that were chosen as pivot rows. Supersteps (4b) and (5b) are the delayed row swap operations, that were delayed in superstep (4) and (5). They are performed using the constructed permutation. Now we need to replicate the elements of $\mathbf{A}_{21}$ and $\mathbf{A}_{12}$ to all nodes that need it to compute the update of the elements in $\mathbf{A}_{22}$ that they own. Note that we already communicated $\mathbf{A}_{21}$ during stages $k_0 \leq k < k_1$. We can store it in a temporary copy, $\mathbf{L}_{21}$. The broadcast that was delayed in superstep (6) is done in superstep (6b). Note that if the block size b provides a balanced number of rows and columns, that is, both M and N divide b, then there is no need for a two-phase broadcast here. As rows may have been swapped locally in $\mathbf{A}_{21}$, those are not yet swapped in the temporary copy $\mathbf{L}_{21}$ that was updated in superstep (0'). Luckily, we know which rows were affected, by looking at the permutation, and there are at most b rows of $\mathbf{A}_{21}$ affected by such swaps, probably less. We can re-broadcast them within our processor column to fix them in $\mathbf{L}_{21}$, along with broadcasting the rows of $\mathbf{A}_{12}$ to every node that needs them, as shown in superstep (4b*). We are still not ready to update $\mathbf{A}_{22}$ now, as we have not yet performed the same row subtractions on $\mathbf{A}_{12}$ and $\mathbf{U}_{12}$ that we performed in $\mathbf{A}_{11}$. These are dependent on the order in which they were performed in $\mathbf{A}_{11}$ for $k_0 \leq k < k_1$, so we could either perform b supersteps for this, or we can redundantly compute it on each node that needs the value, also in b steps, but without extra synchronisation. As every node needs all of the b elements in the columns that it owns, this is the preferred approach. We do however need to replicate the lower-triangle of $\mathbf{A}_{11}$ to all processors, as $\mathbf{L}_{11}$, in order to compute this. This sub-matrix only contains approximately $b^2/2$ elements in total. $\mathbf{L}_{11}$ is constructed in two stages: in superstep (0'), the nodes store the columns of $\mathbf{L}_{11}$ that they own in $\mathbf{A}_{11}$, row-wise for every stage k. Then in superstep (4b*), we

complete the matrix by broadcasting the columns to nodes in the same row. The naming of superstep (4b*) is chosen, because the communication happens in the other subset-environment. We can postpone the receiving of superstep (4b) and (4b*) to merged superstep (5b). Moreover, in case we do not split into subset-environments for node rows and columns, (4b) and (4b*) can be merged into a global superstep.

Clearly, the most computationally intensive part of the algorithm, is the update of $\mathbf{A}_{12}$ and $\mathbf{A}_{22}$. We can define algorithms for this that are parametrised by the available sub-environment, so in case there is a shared-memory sub-environment, we can further divide the work, as shown in Algorithm 7.9 and Algorithm 7.11.

Algorithm 7.7 (continued)

{We continue the algorithm inside the loop iterating k_0.}
{Reconstruct delayed swaps} ▷ superstep (0'')
reconstruct permutation $\Pi(r) : Q \mapsto Q$ from Δ
{Perform delayed swaps} ▷ superstep (4b)
for all $(r, \Pi(r) : (k, r) \in \Delta \wedge r \mod M = s$ **do**
  send $(\Pi(r), 0, \vec{a}_r(0, k_0))$ to $P_t(\Pi(r) \mod M)$ into S_{dswap}
  send $(\Pi(r), k_1, \vec{a}_r(k_1, n)$ to $P_t(\Pi(r) \mod M)$ into S_{dswap}
{Fix rows of $\mathbf{L}_{21}$, complete $\mathbf{L}_{11}$} ▷ superstep (4b*)
for all $(k, r) \in \Delta \wedge k \mod M = s \wedge k \geq k_1$ **do**
  broadcast $(k, \vec{a}_k(k_0, k_1))$ to $P_s(*)$ into $S_{\mathbf{L}_{21}}$
for all $k \in K_0(k_0, k_1) \wedge k \mod N = t$ **do**
  broadcast $(k, \mathbf{L}_{11}(*, k))$ to $P_s(*)$ into $S_{\mathbf{L}_{11}}$
{Store the received rows} ▷ superstep (5b)
for all $(k, j_0, \vec{a}_k) \in S_{\text{dswap}}$ **do**
  write $\vec{a}_k$ into row k of $\mathbf{A}$, starting at column j_0
{Fix the affected rows of $\mathbf{L}_{21}$}
for all $(k, \vec{l_k}) \in S_{\mathbf{L}_{21}}$ **do**
  store $\vec{l_k}$ in row k of $\mathbf{L}_{21}$
{Complete the data in $\mathbf{L}_{11}$}
for all $(k, \vec{l_k}) \in S_{\mathbf{L}_{11}}$ **do**
  store $\vec{l_k}$ in column k of $\mathbf{L}_{11}$
{Broadcast rows of $\mathbf{U}_{12}$} ▷ superstep (6b)
for all $k \in K_0(k_0, k_1) \wedge k \mod M = s$ **do**
  broadcast $(k, \vec{a}_k(k_1, n))$ to $P_t(*)$ into $S_{\mathbf{U}_{12}}$
{Store the received rows} ▷ superstep (0''')
for all $(k, \vec{a}_k) \in S_{\mathbf{U}_{12}}$ **do**
  store $\vec{a}_k$ in $\mathbf{U}_{12}$ in row k
{Update $\mathbf{U}_{12}$ with the subtractions performed in $\mathbf{L}_{11}$}
$\mathbf{U}_{12}(k, *) \leftarrow$ `SubtractRows<`$E_1, E...$`>`$(\mathbf{U}_{12}(k, *), \mathbf{L}_{11}, \mathbf{U}_{12})$
for all $k \in K_0(k_0, k_1) \wedge k \mod M = s$ **do**
  $\vec{a}_k(k_1, n) := \mathbf{U}_{12}(k, *)$

$\mathbf{A}_{22} \leftarrow$ `MatrixUpdate<`$E_1, E...$`>`$(\mathbf{A}_{22}, \mathbf{L}_{21}, \mathbf{U}_{12})$

Algorithm 7.8. `SubtractRows` for $E_1 =$ `NoBSP`.

input : $\mathbf{L}_{11} : (k_1 - k_0) \times (k_1 - k_0)$ matrix,
$\mathbf{U}_{12} : (k_1 - k_0) \times \left\lceil \frac{n - k_1}{N} \right\rceil$ matrix,
output : $\mathbf{U}_{12} : (k_1 - k_0) \times \left\lceil \frac{n - k_1}{N} \right\rceil$ matrix,
updated with row subtractions according to $\mathbf{L}_{11}$.

for $k_0 + 1 \leq k < k_1$ **do**
  for all $i \in K_0(k_0, k)$ **do**
    $\mathbf{U}_{12}(k, *) := \mathbf{U}_{12}(k, *) - \mathbf{L}_{11}(k, i) \cdot \mathbf{U}_{12}(i, *)$

Algorithm 7.9. `SubtractRows` for $E_1 =$ `SharedMemoryBSP`.

There is only a slight modification we need to make here compared to Algorithm 7.8; instead of updating all columns of $\mathbf{U}_{12}$, we divide the columns blockwise over the processors. Since all processors have access to all data, and there are no dependencies, we can simply update the assigned columns. A cyclic distribution is not recommended here, as processors will be working close to each other, triggering the cache-coherency mechanism.

Now that we have the full algorithm and the accompanying sub-algorithms, we can construct the analytic cost for the algorithm. We again have subset synchronisation in this algorithm, so there will be $g_0^{(s)}, g_0^{(t)}, l_0^{(s)}, l_0^{(t)}$ terms. To simplify matters, we will begin with taking the upperbounds for these parameters, so

$$g_0^* := \max\left(g_0^{(s)}, g_0^{(t)}\right), \tag{7.22}$$

$$l_0^* := \max\left(l_0^{(s)}, l_0^{(t)}\right). \tag{7.23}$$

We could later easily separate them, but this simplifies analysis. We will also parametrise the number of rows and columns that are owned by a node:

$$R(k, n) := \max_{0 \leq s < M} |\{i : k \leq i < n \wedge i \mod M = s\}| \approx \left\lceil \frac{n - k}{M} \right\rceil, \tag{7.24}$$

$$C(k, n) := \max_{0 \leq t < N} |\{j : k \leq j < n \wedge j \mod N = t\}| \approx \left\lceil \frac{n - k}{N} \right\rceil, \tag{7.25}$$

Algorithm 7.10. `MatrixUpdate` for $E_1 = \texttt{NoBSP}$.

$$\begin{aligned}
\textit{input} : \mathbf{L}_{21} &: \left\lceil \frac{n-k_1}{M} \right\rceil \times (k_1 - k_0) \text{ matrix}, \\
\mathbf{U}_{12} &: (k_1 - k_0) \times \left\lceil \frac{n-k_1}{N} \right\rceil \text{ matrix}, \\
\mathbf{A}_{22} &: \left\lceil \frac{n-k_1}{M} \right\rceil \times \left\lceil \frac{n-k_1}{N} \right\rceil \text{ matrix}, \\
\textit{output} : \mathbf{A}_{22} &: \left\lceil \frac{n-k_1}{M} \right\rceil \times \left\lceil \frac{n-k_1}{N} \right\rceil \text{ matrix}, \\
&\text{updated for stages } k_0 \leq k < k_1.
\end{aligned}$$

$\mathbf{A}_{22} := \mathbf{A}_{22} - \mathbf{L}_{21}\mathbf{U}_{12}$

Algorithm 7.11. `MatrixUpdate` for $E_1 = \texttt{SharedMemoryBSP}$.

There is again little to be changed here compared to Algorithm 7.10. Similar to Algorithm 7.9, we divide $\mathbf{A}_{22}$ either row- or column-blockwise (depending on data layout), and update the parts assigned to us with the data that is available in shared memory.

where $R(k,n)$ is the number of rows owned by a node between k, n, and $C(k,n)$ the number of columns owned by a node. The cost per superstep is shown in Table 7.1. We neglect small computational parts of the cost. In superstep (0), we need to search $R(k,n)$ elements for the pivot element. In superstep (1), we communicate 2 data words in the processor column: the row and the value of the pivot element. In superstep (2) we divide $R(k,n)$ elements by the pivot element. In superstep (3) we communicate the index of the pivot row in the processor row. In superstep (4) we perform the swap of rows (k,r), so (at most) two processor rows will be communicating the elements they own to the node in the same processor column owning the other row, so a total of $C(k_0,k_1)$ data words are communicated. Superstep (5) is only bookkeeping where the data has to go, and is actually the continuation of superstep (4). In superstep (6), we broadcast the part of the row that we own inside the block, contributing $C(k_0,k_1)(M-1)$ data words, as well as all the elements of column k between row k and n that we own, contributing $R(k+1,n)(N-1)$ data words. Note that this is the superstep where we can reduce the cost with a two-phase broadcast. Superstep (0') updates the remaining part of $\mathbf{A}_{11}, \mathbf{A}_{21}$ with the row and column that were just, with $2C(k+1,k_1)R(k+1,n)$ operations. These supersteps are performed for every $0 \leq k < n$.

Then there are the supersteps containing the work that was delayed. In superstep (0''), there are only a few operations needed to reconstruct the permutation. In superstep (4b), every node owns on

superstep	cost
(0)	$l_0^* + R(k,n)$
(1)	$l_0^* + 2g_0^* \cdot (M-1)$
(2)	$l_0^* + R(k,n)$
(3)	$l_0^* + g_0^* \cdot (N-1)$
(4)	$l_0^* + g_0^* \cdot C(k_0,k_1)$
(5)	l_0^*
(6)	$2l_0^* + g_0^* \cdot C(k_0,k_1)(M-1) + g_0^* \cdot R(k+1,n)(N-1)$
(0')	$2C(k+1,k_1)R(k+1,n)$
(0'')	l_0^*
(4b)	$l_0^* + g_0^* \cdot 2\frac{b}{M} \cdot (C(0,k_0) + C(k_1+1,n))$
(4b*)	$l_0^* + g_0^* \cdot \left(\frac{b}{M} \cdot C(k_0,k_1)(N-1) + C(k_0,k_1) \cdot b \cdot (N-1)\right)$
(5b)	l_0^*
(6b)	$l_0^* + g_0^* \cdot R(k_0,k_1)C(k_1+1,n)(M-1)$
(0''')	$T_{\texttt{SubtractRows}}\texttt{<}E_1, E...\texttt{>} + T_{\texttt{MatrixUpdate}}\texttt{<}E_1, E...\texttt{>}$

Table 7.1: Cost analysis per superstep for Algorithm 7.7. Note that this is an analysis for a single time the superstep is performed.

average $2\frac{b}{M}$ of the rows participating in the swaps, so needs to send $2\frac{b}{M} \cdot (C(0,k_0) + C(k_1+1,n))$ data words, the part left of the current block and the part right of the current block. In superstep (4b*), we repair the rows of $\mathbf{L}_{21}$ that were not properly updated during the initial part. There are at most b such rows, divided over M nodes, so $\frac{b}{M} \cdot C(k_0,k_1)(N-1)$ data words because they need to be rebroadcast. Then there are also the columns of $\mathbf{L}_{11}$ that we need to broadcast so that every node has a full copy of $\mathbf{L}_{11}$, so another $C(k_0,k_1) \cdot b \cdot (N-1)$ data words due to the broadcast. Then superstep (5b) is for bookkeeping again. In superstep (6b), we broadcast the rows of $\mathbf{U}_{12}$ that we own to nodes in the same processor column, contributing to $R(k_0,k_1)C(k_1+1,n)$ data words. Note that due to delaying the communication, this is now (nearly) balanced, as opposed to superstep (6), thus a two-phase broadcast would not improve the cost here.

Finally, we defer the bulk of the work, updating $\mathbf{U}_{12}$ according to $\mathbf{L}_{11}$, and updating $\mathbf{A}_{22}$, to the sub-environment. In case we have no BSP sub-environment, that is, we have to update it sequentially, so then

$$T_{\texttt{SubtractRows7.8}}\texttt{<NoBSP}, E...\texttt{>} := 2\frac{b^2}{2}C(k_1+1,n), \tag{7.26}$$

$$T_{\texttt{MatrixUpdate7.10}}\texttt{<NoBSP}, E...\texttt{>} := 2R(k_1+1,n) \cdot b \cdot C(k_1+1,n). \tag{7.27}$$

As we already established in Algorithm 7.9 and Algorithm 7.11, the work is divisible over the processors in a shared-memory sub-environment without communication and only a single synchronisation, so to obtain the cost for the shared-memory variant, we can just divide the sequential cost by the number of processors in the sub-environment, and add l_1 for the synchronisation cost. The supersteps that are delayed are performed

after each block of size b, so there are $\lceil n/b \rceil$ such supersteps.

We will again analyse the cost for the scenario that we have the environment hierarchy $[E_0, \texttt{SharedMemoryBSP}, \texttt{NoBSP}]$. By the analysis of Bisseling in [Bis04], the choice $M = N = \sqrt{p}$ is optimal for the original algorithm. As the amount of communication and computation has not changed that much, we will adopt this in this analysis as well, as it also simplifies the analysis. By a lengthy analysis with multiple intermediate results, Bisseling condensed the cost of the his LU decomposition algorithm to the dominating terms, which were

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

Instead of reproducing the same analysis with slight modifications for the delayed work, we will analyse where the terms came from, and modify it to account for the delayed work. The original analysis can be found in the work of Bisseling, in chapter 2 of [Bis04]. The term $\frac{2n^3}{3p}$ stems from the matrix update of the remaining matrix. We split that update into two steps: an update of $\mathbf{A}_{11}, \mathbf{A}_{21}$, and the update of both $\mathbf{A}_{12}, \mathbf{A}_{22}$. Roughly speaking, this would split the term into $\frac{2n^2 b}{3p}$ and $\frac{2n^2(n-b)}{3p}$. Note that for now, we only further parallelised the $\frac{2n^2(n-b)}{3p}$ term, thus in terms of our new parameters, this term would translate, in terms of dominating terms, to approximately $\frac{2n^2 b}{3p_0} + \frac{2n^2(n-b)}{3p_0 p_1}$. The $\frac{3n^2}{\sqrt{p}}$ term stems from the search and update of L, which we have not (yet) parallelised, thus remains $\frac{3n^2}{\sqrt{p_0}}$ in terms of our parameters. The $\frac{3n^2 g}{\sqrt{p}}$ stems from the row swap, the row broadcast and the column broadcast. Note however that this relies on a two-phase broadcast, which is not yet implemented in the high-performance algorithm. The row swap and broadcast on the other hand, is balanced due to all the processors owning approximately the same number of rows in the block, and delaying the bulk of the communication involved. Translating it to our parameters, taking into account the unbalanced column broadcast, this term would become $\frac{(2+N)n^2 g_0^*}{\sqrt{p_0}}$. The number of synchronisations for the inner loop remains the same, so the $8nl$ carries over to $8nl_0^*$, but we do need to add the synchronisation for the delayed supersteps, contributing another $5\frac{n}{b}l_0^*$. We can now approximate the cost of the high-performance algorithm, in term of dominating factors, with

$$T_{\texttt{HP-LU}} \approx \frac{2n^2(n-b)}{3p_0 p_1} + \frac{2n^2 b}{3p_0} + \frac{(2+N)n^2 g_0^*}{\sqrt{p_0}} + \left(8n + 5\frac{n}{b}\right) l_0^*. \tag{7.28}$$

The cost for a sequential LU decomposition is approximately $\frac{2n^3}{3}$, with a term of $O(n^2)$ for the partial pivoting. We can see that the bulk of the work is now indeed divided by the product $p_0 p_1$, and the work for the pivoting by $\sqrt{p_0}$ because it is done cooperatively by the nodes in a processor column. Hierarchical parallelisation of the pivot stage is harder to predict, as the data that is processed in each stage k is relatively small. In a perfect world, we could reduce the other two computation terms by a factor p_1 as well, introducing only another nl_1 for synchronisation.

8 Experimental Results

Now that we have theoretically analysed two algorithms, the three-dimensional matrix multiplication, and the High-Performance LU decomposition, we can test out the experimental performance of both algorithms, and see if it corresponds to our expectations. In order to get a more stable result, we perform multiple runs for each configuration that we have, and take the minimum time as representative of the peak performance we achieved. All experiments are performed on Cartesius.

In order to approach the peak performance of the machine, we ideally want to make use of optimised linear algebra routines as part of our algorithm. The implementation of both algorithms was done using the Intel Math Kernel Library (MKL) to handle for example multiplication of sub-matrices. MKL is a collection of optimised linear algebra libraries containing BLAS and LAPACK, developed and optimised by Intel, the manufacturer of the processors that we use in Cartesius. It also contains a shared-memory parallel version of the routines, using Intel Threading Building Blocks (TBB).

8.1 Matrix-Matrix multiplication

For the 3D matrix multiplication, we will be experimenting with square matrices. With the communication pattern in the 3D matrix multiplication we would expect a balanced configuration to perform the best, as this minimises the communication volume. We consider a balanced configuration to be a processor cube where the dimensions are as balanced as possible, taking into account the constraints of the possible divisors of the number of processors. However, we will be dealing with inter-node and inter-socket communication within the node, so the actual optimal configuration may slightly differ from a balanced configuration.

8.1.1 Initial comparison

In order to get an initial indication of performance of different environment configurations, we perform a measurement in which we utilise a full node, with a fixed processor cube dimension of $3 \times 2 \times 4$, and compare the sequential computation time (with sequential BLAS) to our parallel

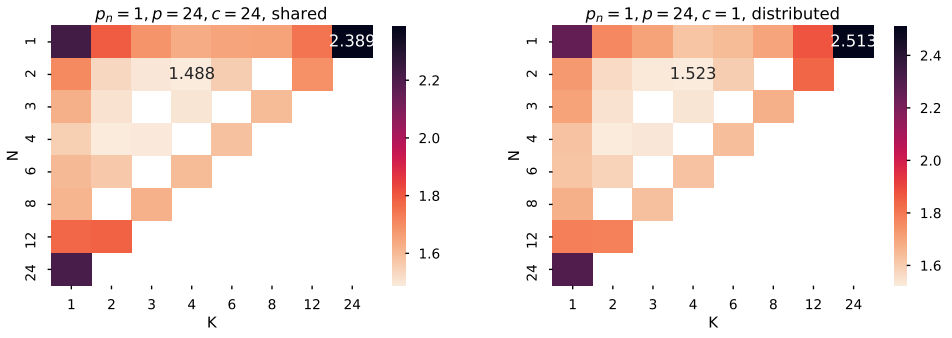
n	Sequential	Shared	Distributed	MKL-TBB
4096	3.136s	0.356s	0.404s	0.290s
8192	24.63s	1.488s	1.523s	1.332s

Table 8.1: Comparison of the time a matrix multiplication of square matrices of size $n \times n$ takes on a single, fully utilised physical node.

algorithm both on a shared-memory backend, and on a distributed memory environment, and finally a parallel BLAS implementation with MKL. The results are shown in Table 8.1. We notice that most of the times are not that far apart, but the MKL-TBB (thus using parallel BLAS) performs the best in both cases. While this is a parallel implementation, it is focussed around shared-memory parallelism, so in case we want to utilise multiple nodes, we will be needing a distributed-memory BSP main environment. This combination would thus seem the most promising.

8.1.2 Varying the processor cube

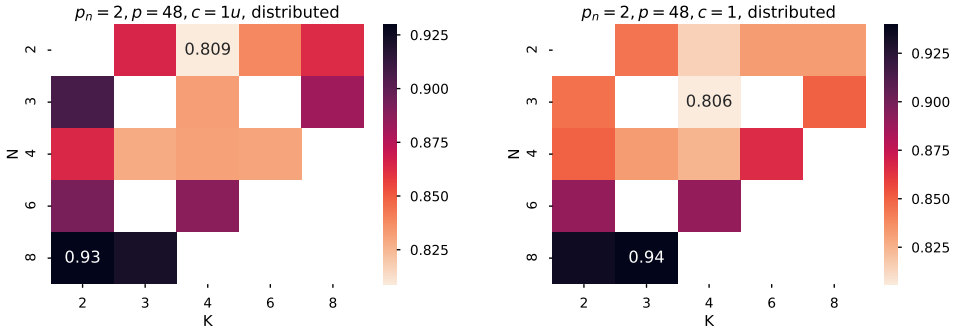
The next thing we will measure is what happens when we vary the processor cube dimensions. We choose a fixed number of algorithmic nodes $p = 24$, and try different configurations of the processor cube. We represent the best time achieved in a heatmap, as shown in Figure 8.1. The axis of the heatmap contain two dimensions of the processor cube, and the third dimension can easily be derived. When we look at the heatmaps for the shared-memory and distributed-memory version, the pattern is almost identical. However, shared-memory has a slight edge in terms of time, due to overhead from the communication and synchronisation on



(a) Shared memory with one core per process.

(b) Distributed memory with one core per process.

Figure 8.1: Comparison of different processor cube configurations for the 3D matrix multiplication with $p = 24$ on $p_n = 1$ physical node. The measurement is performed both on shared memory and distributed memory. The matrices were all of size 8192×8192. Time is measured in seconds (see the colorbar). The minimum and maximum time are shown in the corresponding cells.



(a) Distributed memory with global synchronisation.

(b) Distributed memory with subset synchronisation.

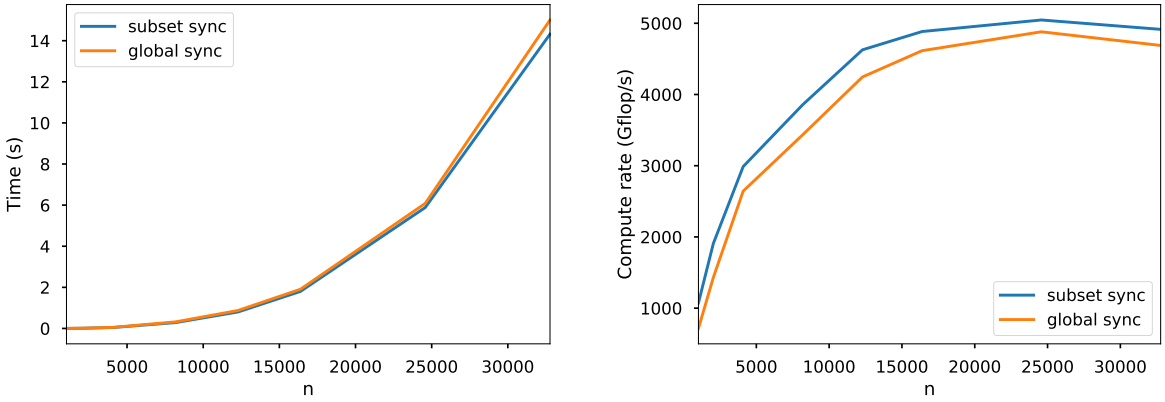
Figure 8.2: Comparison of different processor cube configurations for the 3D matrix multiplication with $p = 48$. The measurement is performed with global synchronisation and subset synchronisation. The matrices were all of size 8192×8192. Time is measured in seconds.

distributed-memory. Indeed, the best times are achieved for balanced configurations. There seems to be a slight bias towards smaller N. This could be due to the existence of sockets, where $N = 2$ would minimise the inter-socket communication.

We can scale this up to $p = 48$, but then we lose the shared-memory comparison. We can however also compare the global and subset synchronisation. This comparison is shown in Figure 8.2. The most balanced configuration would have two processor cube dimensions equal to 4, and one dimension equal to 3. However, if we look at Figure 8.2a, we notice that $N \times K \times M = 2 \times 4 \times 6$ gives the best performance. This could be due to the fact that $N = 2$ now minimises the inter-node communication. But if we look at Figure 8.2b, the balanced configuration $3 \times 4 \times 4$ now has the advantage. This is now probably due to the reduced synchronisation cost; $2 \times 4 \times 6$ is still one of the better configurations, but due to the reduced synchronisation, we can now move to a more balanced configuration. The times for the split and global synchronisation versions are very similar in the best and worst case: this is due to the fact that there is a very minimal number of supersteps, so the effect of reduced synchronisation cost is not that significant.

8.1.3 Varying the matrix size

The next experiment we perform is with a fixed, balanced processor cube, consisting of distributed memory nodes, each with 24 cores in the sub-environment. For the sub-environment, an optimised parallel BLAS implementation was chosen to handle the multiplication of blocks of the matrix, as mentioned in Section 8.1.1. The measured times are shown in Figure 8.3. The compute rate is computed by dividing $2n^3$ by the time it takes to perform the parallel matrix multiplication. Of



(a) Measured time for the matrix multiplication.

(b) Compute rate achieved during the matrix multiplication.

Figure 8.3: Comparison of different matrix sizes for the 3D matrix multiplication with $N \times K \times M = 2 \times 2 \times 2 = 8$ nodes, with 24 cores each, utilised by an optimised parallel BLAS routine. The measurement is performed with global synchronisation and subset synchronisation.

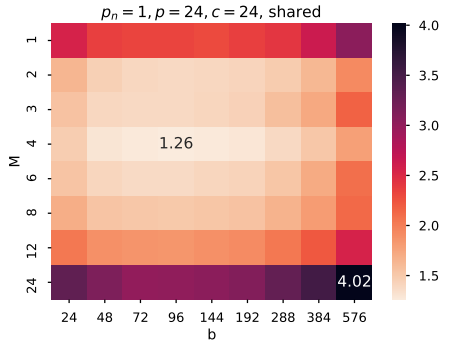
course, there is much more work done in terms of communication and synchronisation, and some extra work had to be done to recombine the results, but $2n^3$ is the amount of work that needed to be done in the first place. What we immediately notice is that the measurements follow the same curve, as expected. What is interesting to see, is that although there is a fixed number of supersteps in the algorithm, the discrepancy in time grows between the subset synchronisation and global synchronisation. This means that not only the synchronisation cost is decreased, but also the communication cost. This confirms our expectation that reducing congestion can also reduce communication cost. What we can also notice is that the compute rate is already past its peak for this experiment, while the difference in communication volume and computation should only grow, thus stabilizing the compute rate. However, there are other factors at play here as well, like slower memory access because of large matrices.

If we look at the peak performance, the global synchronisation reaches 4880 Gflop/s, while the subset synchronisation reaches 5046 Gflop/s. This is an easy performance gain on a communication pattern that is already there. As we can see, the compute rate per core is much larger than the r measured in the benchmark. However, the cost does follow the predicted shape of the dominating $O(\frac{2n^3}{p_0 p_1})$ term.

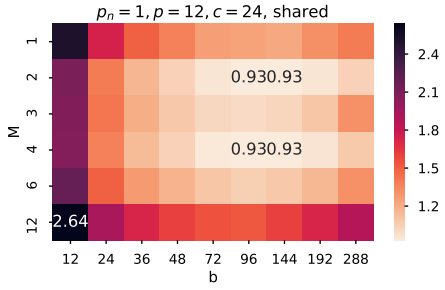
8.2 LU decomposition

For the LU decomposition, several scenarios were implemented and measured. First, we will try to determine an optimal combination of processor matrix dimension $M \times N$, and block size b, for a given matrix size n, and

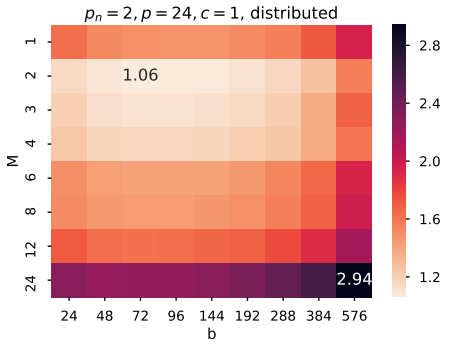
several node configurations. Then for the best combination $M \times N$ and b and node configuration, we will vary the matrix size, making sure we also check for variations in block size, as varying matrix size may change the optimal block size b. We will be using a matrix that is also used in the accompanying educational software package BSPedupack to [Bis04] by Bisseling. The matrix is the product of specific matrices $\mathbf{L}, \mathbf{U}$, and then row-rotated such that the last row of the matrix is moved to the first row. Here $\mathbf{L}$ is lower triangular, and contains all $\frac{1}{2}$ entries, except for the diagonal of course, which is 1, and $\mathbf{U}$ is upper triangular, with all entries equal to 1. Without the row rotation, there would be no swaps needed for the LU decomposition, the pivot element would always be on the diagonal. By performing the row rotation, the row needs to be swapped with the next row at every stage k.



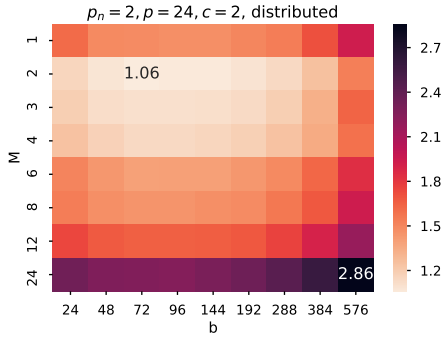
(a) Shared memory with one core per process.



(b) Shared memory with two cores per process.



(c) Distributed memory with one core per process.



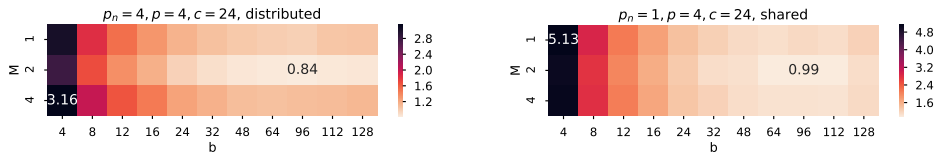
(d) Distributed memory with two cores per process.

Figure 8.4: Comparison of several configurations on either one or two physical nodes. The total number of processors (the number of algorithmic nodes) is p, whereas p_n is the number of physical nodes. In case of a shared memory environment, c is the total number of cores available. In case of a distributed memory improvement, c is the number of cores available to each algorithmic node. Time is measured in seconds.

8.2.1 Varying the processor matrix and block size

We choose a fixed size matrix $\mathbf{A}$ of size 4096×4096. We utilise subset synchronisation to reduce synchronisation time. The results are presented in a heatmap, where the horizontal axis represents the block size, and the vertical axis represents M from the processor matrix. N can be derived by dividing p by M, as $p = MN$. The title inside the figure states the number of physical nodes p_n, the number of processors per node c and the number of algorithmic nodes p. In case the title states "shared", then we divide c/p to obtain the number of cores nested inside a shared memory node. For example Figure 8.4a was performed in a shared-memory main environment, with one core per process. In case the title states "distributed", then every algorithmic node has c cores at its disposal. For example Figure 8.4d was performed in a distributed memory main environment, with two cores per process. In these experiments, we leverage parallel BLAS via MKL; thus the extra cores are used by the parallel MKL implementation. As we can see from Figure 8.4, the shared memory variant certainly has an advantage from using parallel MKL. This is probably due to (at least) two factors: there are less participants in the synchronisation, and the node is divided into two sockets that share memory. Inter-socket memory access is slower than intra-socket memory access, and thus most processes will only work intra-socket. The threads from the extra socket are only utilised by the parallel MKL routines, and only as much as is beneficial. When using two physical nodes, the extra core per process does not increase performance much, only slightly in the worst-case scenarios.

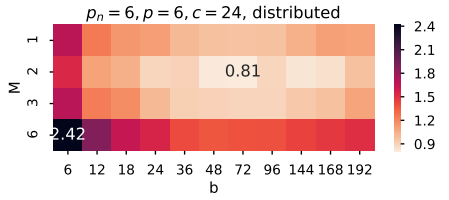
As a variant of this experiment, a comparison is made by choosing a fixed size for the main environment, and either using a shared-memory environment, limiting the parallel MKL threads, or using a distributed-memory environment, where each node has full access to 24 parallel MKL threads. The results are shown in Figure 8.5 and Figure 8.6 for $p \in \{4, 6, 12\}$ nodes. As we can see, the distributed-memory environment mostly has a slight edge over the shared-memory environment, where only the configuration $p = 12, M = 12$ is worse due to imbalance in



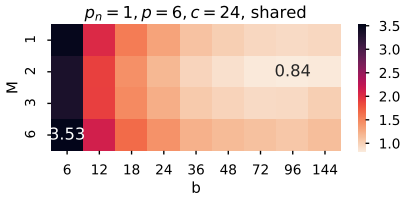
(a) Four distributed memory nodes, with parallel MKL over 24 threads each.

(b) Four shared memory nodes, with parallel MKL over 6 threads each.

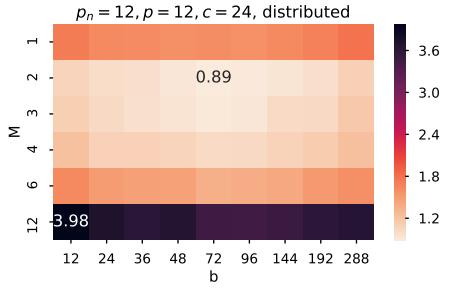
Figure 8.5: Comparison of several configurations on either shared memory (restricting the number of MKL threads) or distributed memory nodes, giving MKL access to the full 24 threads. Time is measured in seconds.



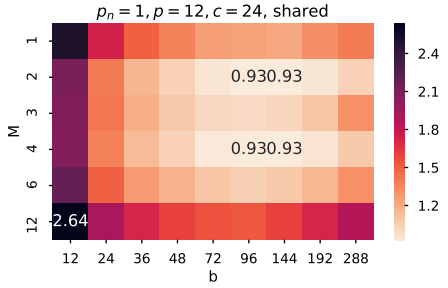
(a) Six distributed memory nodes, with parallel MKL over 24 threads each.



(b) Six shared memory nodes, with parallel MKL over 4 threads each.



(c) Twelve distributed memory nodes, with parallel MKL over 24 threads each.



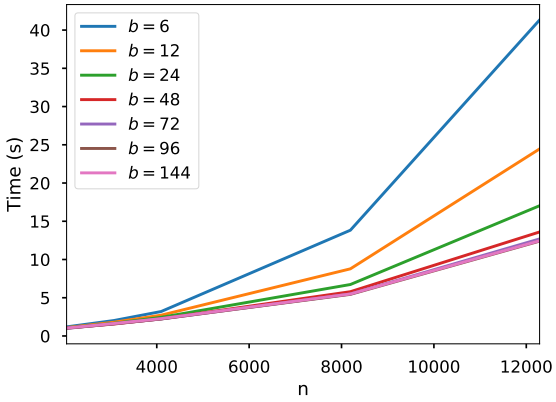
(d) Twelve shared memory nodes, with parallel MKL over 2 threads each.

Figure 8.6: Comparison of several configurations on either shared memory (restricting the number of MKL threads) or distributed memory nodes, giving MKL access to the full 24 threads. Time is measured in seconds.

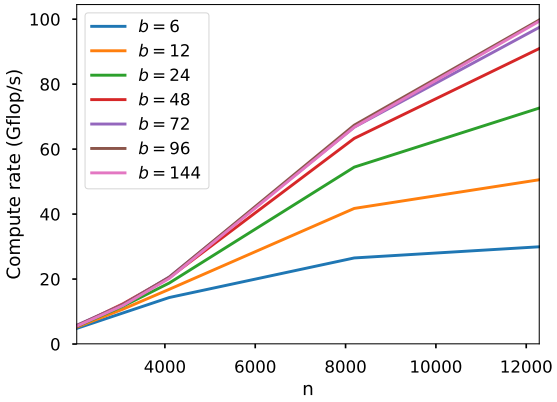
communication induced by the processor matrix of dimensions 12×1. What is also interesting to see, is that $b \in \{72, 96\}$ gives the overall best performance, both in Figure 8.4 and Figures 8.5 and 8.6. The best performance also favours $M < N$, while they should still be somewhat close. This is probably due to the fact that a *two-phase* broadcast was not yet used for the broadcasting of columns of A_{21} during the pivoting phase. This causes an imbalance in communication in the pivoting phase.

8.2.2 Varying matrix size and block size

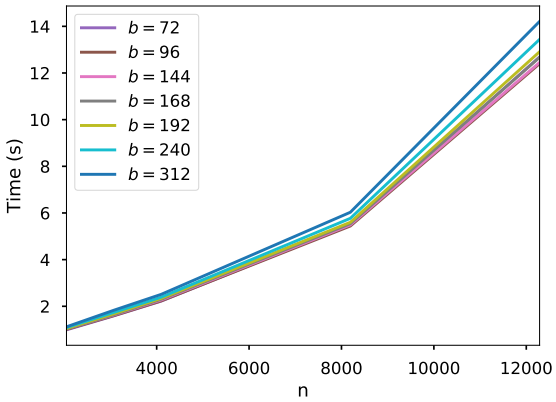
As $p = 6, M = 2$ gave the best performance in Figure 8.5, we will fix this configuration for our next experiment. In this experiment, we will measure the time the LU decomposition takes for increasing matrix size. We will still vary the block size b, as the optimal block size may be different for a matrix of different size. In this experiment, we will also look at the difference in time when we compare global synchronisation to subset synchronisation. We look at both the time the LU decomposition takes, and the computation rate that is achieved if we divide the theoretical number of operations by the time. Recall that the theoretical cost was approximately $\frac{2}{3}n^3$. Figure 8.7 shows the time taken and the compute rate that is achieved when we use global synchronisation everywhere. We can see that $b = 96$ is still close to optimal for all matrix sizes that were



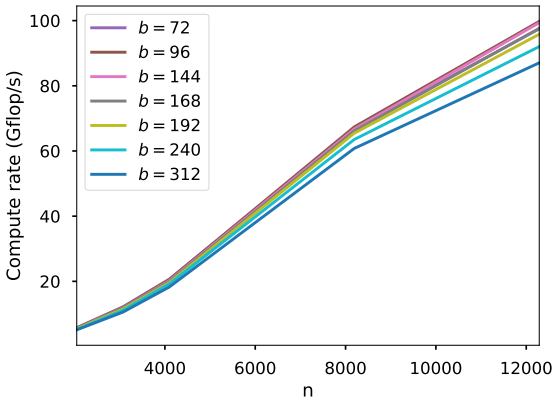
(a) Elapsed time for a LU decomposition of $A : n \times n$



(b) Compute rate for a LU decomposition of $A : n \times n$



(c) Elapsed time for a LU decomposition of $A : n \times n$

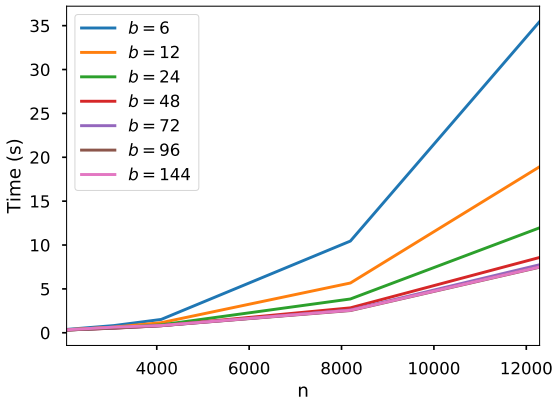


(d) Compute rate for a LU decomposition of $A : n \times n$

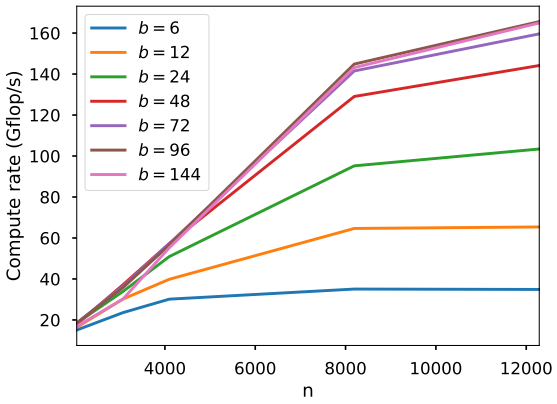
Figure 8.7: Time taken and compute rate for LU decomposition of increasing size n. Synchronisation was performed globally with $p = 2 \times 3$ nodes. The second row in the figure are intended to emphasize the effect of increasing the p beyond $p \in \{72, 96, 144\}$.

tested, while $p \in \{72, 144\}$ are close. A compute rate of slightly under 100 Gflop/s is achieved for the largest matrix, and it still seems increasing as the matrix size increases.

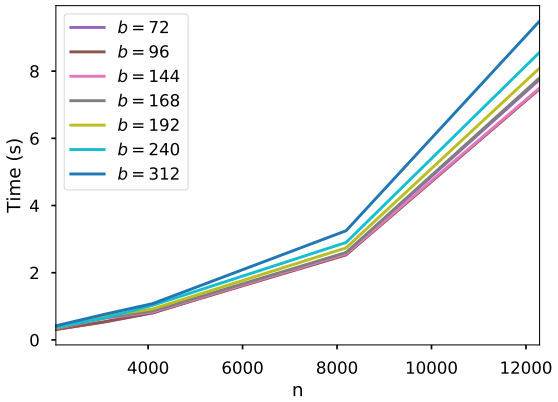
If we compare these measurements for matrix size 12288×12288 to Figure 8.8, we notice that the worst-case time, for $b = 6$ is easily decreased by nearly 6 seconds, while 4.9 seconds are still shaved off from the best time achieved for $b = 96$. This gives a major increase in the compute rate: we now achieve 165 Gflop/s, by just splitting into subsets, while the subset pattern is already there. Performance could easily be increased by implementing the two-phase broadcast.



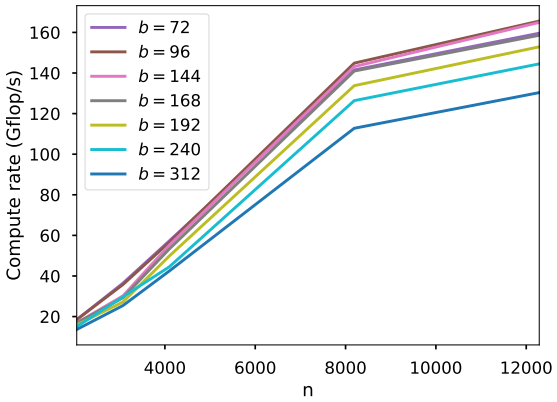
(a) Elapsed time for a LU decomposition of $A : n \times n$



(b) Compute rate for a LU decomposition of $A : n \times n$



(c) Elapsed time for a LU decomposition of $A : n \times n$



(d) Compute rate for a LU decomposition of $A : n \times n$

Figure 8.8: Time taken and compute rate for LU decomposition of increasing size n. Synchronisation was performed in subsets on $p = 2 \times 3$ nodes. The second row in the figure are intended to emphasize the effect of increasing the p beyond $p \in \{72, 96, 144\}$.

BSPedupack			
4×4	6×4	12×12	SyncLib
29.67s	30.03s	26.95s	2.53s

Table 8.2: Time comparison for the LU decomposition of $A : 8192 \times 8192$ for BSPedupack vs SyncLib. SyncLib utilised $p = 2 \times 3$ nodes with 24 cores, so $p = 144$.

We can also compare the performance to the LU decomposition algorithm provided in BSPedupack version 1 using MulticoreBSP version 1.2.0; it uses the same matrix, with the same pivoting strategy, except it does not perform the delayed matrix update in this version. The times for a matrix of size 8192×8192 are shown in Table 8.2. As we can see, increasing the number of processors for the BSPedupack implementation does not scale further with the number of processors, and SyncLib is already a factor 10.6 faster than the BSPedupack implementation, while using the same number of processors. Thus, while the matrix, the number of participating processors and the pivot strategy have not changed, and we even introduced some communication imbalance, we still gained a factor 10 speedup by postponing some work. This indicates that algorithmic changes can lead to major speedup.

Summarising, we obtained a large speedup over the current BSPedupack implementation that was done without the delayed work. The algorithm works well together with the parallel MKL implementation. A direct improvement that can still be made is to utilise a two-phase broadcast during the pivoting phase. While we can clearly see that the cost follows the shape of the dominating term $\frac{3n^3}{p_0 p_1}$, an exact comparison with the theoretical cost is hard, because we have a very mixed combination of flops: some are comparison operations in the pivot search, some division, some matrix multiplication and some subtraction. As we have seen in the benchmark comparison between MulticoreBSP and SyncLib in Chapter 6, even the same theoretical computational work may have very different compute rates in practice. In the future, visualisation of different supersteps with the profiler may provide a better insight in how each superstep contributes to the total time, and then we may be able to better analyse the cost compared to the theoretical cost.

8.3 Compute rate comparison

We have performed two algorithms, with very different compute rate. We can now compare this to the entry in the Top500 [TOP17], by looking at the single-core performance. The maximum reported performance for 38880 cooperating cores was 1088.51 Tflop/s, while the theoretical maximum performance of this collection of processors (without communication or synchronisation) would be 1327.1 Tflop/s. This means a single core would in practice contribute approximately 28 Gflop/s, while in

our experiment with the LU decomposition, the 165 Gflops is achieved by 144 cooperating cores, is only achieving 1.15 Gflop/s, a factor of 24 slower. Although we have not yet reached the full potential compute rate of this implementation (we still see an increasing compute rate), there certainly seems to be room for improvement, the easiest being the two-phase broadcast. This in turn might change the optimal block size b, even further increasing the compute rate. Moreover, only the bulk of the work is done in parallel, while we may still benefit from parallel pivot searching and division of the columns we store in $\mathbf{L}$. On the other hand, if we look at the matrix multiplication, we achieved a compute rate of 5046 Gflop/s with 192 cores, thus approximately 26.3 per core, which extrapolates to 1021 Tflop/s for 38880 cores. This already comes very close to the peak performance of the entry in the Top500.

9 Conclusion

The goal of this thesis was to generalise the BSP model to hierarchical heterogeneous architectures. We started out by analysing the different architectural properties, and identified the distinction between a communication network and shared-memory as the leading property in formalising a model. We have looked at extensions of the BSP model, and took inspiration from these extensions to identify what proved to work and what not. By combining these properties, and making some properties more flexible, we formalise a framework for analysis, as well as the SyncLib library for experimental analysis. The development of SyncLib required significant effort, as it was built from scratch with the new extension in mind. SyncLib is open-sourced on GitHub[1].

We have extended the model with flexible subset-synchronisation through subset-environments, that can coexist throughout the algorithm. By scoping communication and synchronisation to the subset-environment, for pre-existing patterns in the algorithms, there is both an easy reduction in the total runtime, as well as a simplification of communication logic in such structured subset patterns. Examples of such pre-existing patterns are the processor matrix in the LU decomposition algorithm, and the processor cube in the matrix multiplication algorithm. These patterns are very common in linear algebra algorithms. A pairwise benchmark of latency and bandwidth has also been presented, in which the heterogeneous properties of the communication network were clearly visible. Two algorithms for improving this cost for subset-environment patterns were also presented, but due to technical difficulties, this was not yet feasible to implement in the current state of SyncLib, so it has not yet been incorporated into the experiments. Future versions of SyncLib will be extended with these features.

The hierarchical properties of a machine can now be captured and utilised in SyncLib, with the introduction of sub-environments. In the mathematical analysis, this introduces a separation of the algorithm into a main algorithm and sub-algorithms for different sub-environments. The main algorithm will then exist of the inter-node communication, as well as mild computational work, while the sub-algorithms will handle the bulk

[1]Located at https://github.com/Zefiros-Software/SyncLib

of the work. Not only does this separation allow hierarchical algorithms, but also allows for future extension to "exotic" sub-environments, such as a sub-environment utilising the GPU instead of the CPU, or a sub-environment with a different parallel model. While the cost of these sub-environments needs to be made compatible, the cost expression easily allows this, as it has sub-expressions for sub-algorithms.

The addition of the subset-synchronisation and the sub-environments into the cost analysis does not significantly complicate the cost analysis, as we have seen in the analysis of the hierarchical matrix-matrix multiplication and the LU decomposition. However, the addition of optimised subroutines for linear algebra complicates the exact cost analysis in experiments, as these subroutines are highly optimised for cache-usage, and can utilize many optimised operations as opposed to un-optimised operations from a handwritten loop; such operations can act on multiple data words in a single processor instruction, or can sometimes merge two mathematical operations into a single instruction. Still, the dominating term was visible in both the matrix multiplication and the LU decomposition. A profiler visualising the computational cost and communication cost and volume per superstep may greatly help analysis in these cases, so future versions of SyncLib will be extended with a built-in profiler.

Another important part of the thesis was analysing linear algebra algorithms, because they are the building blocks for more complicated algorithms. Therefore, a High-performance version of the LU decomposition was also developed and analysed. Compared to the educational version in BSPedupack accompanying the book of Bisseling [Bis04], when run on the supercomputer Cartesius, there was a huge improvement in speed and scalability of the algorithm. However, it does not yet come close to the peak performance reported in the Top500 LINPACK benchmark for Cartesius. The matrix multiplication did come very close to this entry in the Top500 in terms of single-core performance. This is due to the constant number of supersteps, and (relatively) small communication volume, as well as the utilisation of optimised subroutines for nearly all the work. Using the combination of distributed-memory BSP and optimised parallel subroutines on shared memory proved to be the best combination for performance.

To further summarise the conclusion, some useful extensions have been made to the model, and an accompanying library has been developed. Analysis has not been severely complicated by the extension, and even allows for flexible extension to future architectures. While performance has improved, and some algorithms nearly reach peak performance, other algorithms still need more work to approach the peak performance of the machine. The improvement of these algorithms may be helped by a visual profiler of superstep contributions to the overall cost.

Bibliography

[ACS90] Aggarwal, Alok, Chandra, Ashok K, and Snir, Marc. "Communication complexity of PRAMs". In: *Theoretical Computer Science* vol. 71. no. 1 (1990), pp. 3–28.

[Aga+95] Agarwal, Ramesh C, Balle, Susanne M, Gustavson, Fred G, Joshi, Mahesh, and Palkar, P. "A three-dimensional approach to parallel matrix multiplication". In: *IBM Journal of Research and Development* vol. 39. no. 5 (1995), pp. 575–582.

[BB17] Buurlage, Jan-Willem and Bannink, Tom. *Bulk*. 2017. URL: https://github.com/jwbuurlage/Bulk (visited on 11/07/2017).

[BBB] Buurlage, Jan-Willem, Bannink, Tom, and Bisseling, Rob. "Bulk: a Modern C++ Interface for Bulk-Synchronous Parallel Programs". Accepted for publication in Proceedings EuroPar 2018.

[BBW15] Buurlage, Jan-Willem, Bannink, Tom, and Wits, Abe. *Epiphany BSP*. 2015. URL: http://www.codu.in/ebsp/ (visited on 11/07/2017).

[BBW16] Buurlage, Jan-Willem, Bannink, Tom, and Wits, Abe. "Bulk-synchronous pseudo-streaming algorithms for many-core accelerators". In: *arXiv preprint arXiv:1608.07200* (2016).

[Bil+01] Bilardi, Gianfranco, Fantozzi, Carlo, Pietracaprina, Andrea, and Pucci, Geppino. "On the effectiveness of D-BSP as a bridging model of parallel computation". In: *Computational Science-ICCS 2001* (2001), pp. 579–588.

[Bil+96] Bilardi, Gianfranco, Herley, Kieran T, Pietracaprina, Andrea, Pucci, Geppino, and Spirakis, Paul. "BSP vs LogP". In: *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*. ACM. 1996, pp. 25–32.

[Bis04] Bisseling, Rob H. *Parallel Scientific Computation: A structured approach using BSP and MPI*. Oxford University Press, 2004.

[BM94] Bisseling, Rob H and McColl, William F. "Scientific computing on bulk synchronous parallel architectures". In: *Technology and Foundations: Information Processing '94 Vol I* vol. (ed. Pehrson, B. and Simon, I.), Volume 51 (1994), pp. 509–514.

[Cul+93] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R., and Von Eicken, T. "LogP: Towards a realistic model of parallel computation". In: *ACM Sigplan Notices*. Vol. 28. 7. ACM. 1993, pp. 1–12.

[Cul+96] Culler, D.E., Karp, R.M., Patterson, D., Sahay, A., Santos, E. E, Schauser, K.E., Subramonian, R., and Eicken, T. von. "LogP: A practical model of parallel computation". In: *Communications of the ACM* vol. 39. no. 11 (1996), pp. 78–85.

[DK96] De la Torre, Pilar and Kruskal, Clyde P. "Submachine locality in the bulk synchronous setting". In: *European Conference on Parallel Processing*. Springer. 1996, pp. 352–358.

[DVV17] Duijn, M van, Visscher, Paul E, and Visscher, Koen M. *Zefiros-Software/BSPLib 1.1.11*. Oct. 2017.

[Eul40] Euler, Leonhard. "De summis serierum reciprocarum". In: *Commentarii academiae scientiarum Petropolitanae* vol. 7. no. 1740 (1740), pp. 123–134.

[HDM98] Hill, Jonathan M.D., Donaldson, Stephen R., and McEwan, Alistair. *Installation and User Guide for the Oxford BSP toolset (v1.4) implementation of BSPlib*. Tech. rep. Oxford University Computing Laboratory, 1998.

[HFE10] Hamidouche, K., Falcou, J., and Etiemble, D. "Hybrid bulk synchronous parallelism library for clustered SMP architectures". In: *Proceedings of the fourth international workshop on High-level parallel programming and applications*. ACM. 2010, pp. 55–62.

[Hil+98] Hill, Jonathan MD, McColl, Bill, Stefanescu, Dan C, Goudreau, Mark W, Lang, Kevin, Rao, Satish B, Suel, Torsten, Tsantilas, Thanasis, and Bisseling, Rob H. "BSPlib: The BSP programming library". In: *Parallel Computing* vol. 24. no. 14 (1998), pp. 1947–1980.

[Int17] Intel. *Intel® Threading Building Blocks*. 2017. URL: https://www.threadingbuildingblocks.org/ (visited on 11/07/2017).

[JáJ92] JáJá, Joseph. *An introduction to parallel algorithms*. Vol. 17. Addison-Wesley Reading, 1992.

[JPA03] J., Dongarra Jack, Piotr, Luszczek, and Antoine, Petitet. "The LINPACK Benchmark: past, present and future". In: *Concurrency and Computation: Practice and Experience* vol. 15. no. 9 (2003), pp. 803–820.

[Keß00] Keßler, Christoph W. "NestStep: nested parallelism and virtual shared memory for the BSP model". In: *The Journal of Supercomputing* vol. 17. no. 3 (2000), pp. 245–262.

[KKT01] Keller, Jörg, Kessler, Christoph, and Träff, Jesper. *Practical PRAM programming*. WileyInterscience, J. Wiley & Sons, Inc., 2001.

[KL70] Kernighan, B. W. and Lin, S. "An Efficient Heuristic Procedure for Partitioning Graphs". In: *Bell System Technical Journal* vol. 49. no. 2 (1970), pp. 291–307.

[MPI16] MPI Forum. *MPI Forum*. 2016. URL: http://mpi-forum.org/ (visited on 11/07/2017).

[NVI07] NVIDIA, CUDA. "Compute unified device architecture programming guide". In: (2007).

[Ope17] OpenMP. *The OpenMP API specification for parallel programming*. 2017. URL: http://www.openmp.org/ (visited on 11/07/2017).

[Par14] Parallella. *The Parallella Board*. 2014. URL: https://www.parallella.org/ (visited on 11/07/2017).

[Sui06] Suijlen, Wijnand J. *BSPonMPI*. 2006. URL: http://bsponmpi.sourceforge.net/ (visited on 11/07/2017).

[Ter+08] Terboven, C., Mey, D. an, Schmidl, D., Jin, H., and Reichstein, T. "Data and thread affinity in OpenMP programs". In: *Proceedings of the 2008 workshop on Memory access on future processors: a solved problem?* MAW '08. Ischia, Italy: ACM, 2008, pp. 377–384.

[Tis98] Tiskin, Alexandre. "The bulk-synchronous parallel random access machine". In: *Theoretical Computer Science* vol. 196. no. 1 (1998), pp. 109–130.

[TOP17] TOP500.org. *SURFsara - Cartesius 2*. 2017. URL: https://www.top500.org/system/178551 (visited on 06/01/2018).

[Val11] Valiant, Leslie G. "A bridging model for multi-core computing". In: *Journal of Computer and System Sciences* vol. 77. no. 1 (2011), pp. 154–166.

[Val90] Valiant, Leslie G. "A bridging model for parallel computation". In: *Communications of the ACM* vol. 33. no. 8 (1990), pp. 103–111.

[Wyl79] Wyllie, James C. "The complexity of parallel computations". PhD thesis. Cornell University, 1979.

[Yze14] Yzelman, Albert-Jan. *MulticoreBSP*. 2014. URL: http://www.multicorebsp.com/ (visited on 11/07/2017).