universität wien

MASTERARBEIT

Titel der Masterarbeit

„Implementation of two Broadcast Algorithms on the Intel SCC Architecture“

verfasst von

Markus Alexander Pichler, BSc

angestrebter akademischer Grad

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2013

Studienkennzahl lt. Studienblatt:	A 066 940
Studienrichtung lt. Studienblatt:	Scientific Computing UG2002
Betreuerin / Betreuer:	Dr. Jesper Larsson Träff, MSc PhD

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe.

Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Wien, am 30. Juni, 2013

Markus Pichler

Abstract

In modern parallel scientific computing applications and interfaces broadcasts are one of the most important functions, because they are used in almost all algorithms that use multiple cores. Therefore the implementation of a reliable high performance algorithm is essential. We will discuss two different kinds of algorithms that are mathematically optimal, but differ in the point that one takes care of the underlying hardware and is optimized to it and the other one runs on any kind of hardware. The condition is of course that each processor can communicate with each other processor over a network. We will implement both algorithms on the SCC (Single-Chip Cloud Computer) from Intel. It is a 48 core chip with a matrix like core arrangement. The cores are connected to each other via a network. A big part of our work will also be to investigate the SCC and show its advantages and disadvantages. The algorithm that takes advantage of the underlying hardware knows that the cores are aligned as a matrix and therefore can optimize the communication that only cores that are neighbors communicate with each other. The advantage here is that the latencies due to the underlying network get minimized. We want to do a lot of performance comparisons between the two algorithms to see if it is necessary to especially adapt the broadcast algorithm to the hardware, or if the performance gap is not that big that it is not worth the effort to implement an algorithm that is especially designed for a given hardware. The algorithm that uses the hardware setup may get a little bit handicapped when running on the SCC compared to something like a cluster, because all cores are on the same chip and therefore the latencies are very low in general, but we still expect him to outperform the general approach.

Zusammenfassung

In modernen, parallelen Scientific Computing Anwendungen und Schnittstellen sind Broadcasts eine der wichtigsten Funktionalitäten. Sie sind essenziell in fast allen Algorithmen die eine Vielzahl an Prozessoren verwenden. Aufgrund dieser Abhängigkeit ist die Implementierung eines verlässlichen und performanten Algorithmus besonders wichtig. Wir werden zwei verschiedene Broadcast Algorithmen präsentieren, die beide mathematisch optimale Laufzeiten haben, aber sich darin unterscheiden, dass einer auf ein bestimmtes Hardware Layout zugeschnitten ist und der andere auf einem beliebigen Layout läuft. Die Einschränkung hierbei ist, dass jeder Prozessor zumindest mit jedem kommunizieren kann. Wir werden beide Algorithmen auf dem SCC (Single-Chip Cloud Computer) von Intel implementieren. Die Besonderheit des SCC Prozessors ist, dass er 48 Kerne auf einem Chip hat, die in einer Matrix angeordnet sind. Die Kerne sind über ein Netzwerk miteinander verbunden. Ein Großteil unserer Arbeit wird darin bestehen die Vor– und Nachteile des SCCs aufzuzeigen. Der Broadcast Algorithmus, der Rücksicht auf das zugrunde liegende Hardware Layout nimmt, weiß dass die Kerne matrixförmig angeordnet sind und kann die Kommunikation so optimieren, dass nur Kerne die benachbart sind miteinander kommunizieren müssen. Dies wiederum minimiert die Netzwerk Latenzzeiten. Wir werden die Performanz der beiden Algorithmen untersuchen und beurteilen ob es sinnvoll ist einen Broadcast Algorithmus zuzuschneiden, oder ob es den Aufwand nicht wert ist und ein allgemeiner Algorithmus zufriedenstellende Ergebnisse aufweisen kann. Dadurch dass die Implementierung und Messung auf dem SCC erfolgt, hat der allgemeine Algorithmus wohl einen Vorteil, da die Latenzzeiten sehr niedrig sind, da sich alle Kerne auf einem Chip befinden, im Gegensatz zu einer Implementierung auf einem Cluster, der weitaus größere Latenzzeiten aufweist. Dennoch erwarten wir, dass der maßgeschneiderte Algorithmus den Allgemeinen schlägt.

1 SCC – Single-Chip Cloud Computer

1.1 Introduction

The Single-Chip Cloud Computer is a research chip from Intel that was designed as the second part of the Intel Tera – scale Program [1]. The Tera – scale program was announced in February in 2007 and was intended to explore today's state of the art multi-core systems by scaling up the number of cores to 100s of cores. With that many cores the application developers will be forced to create their programs with much more parallelism rather than seriality. This leads to a change of the general programming model. Five years later it can be seen as common. The goals of the program were to increase the performance and the energy – efficiency by increasing the number of cores and slowing down the individual core frequency. Intel first designed an 80 core non – IA (Intel Architecture) chip [2] that broke the 1 Teraflop/s wall. The second chip that was created in the Tera – scale program is the Single-Chip Cloud Computer with 48 cores.

1.2 Research Goals

This section is based on [3]. Intel's main goals behind the project are to study the "programmability and scalability" of a "shared memory message – passing architecture" with many cores on one chip. The results are important for the many – core chip development in the near future to gain a feeling how hard it is to get programs to run and even more harder to get some reasonable performance out of it. Another aspect of the architecture is that the single cores are arranged together as a 2D matrix. They are connected to each other in this way via a network. This setting gives new opportunities for algorithms, because the access time from a core to one of its neighbors should be significantly shorter than the time to a more distant one. This will be discussed in the later chapters. The SCC chip also has a "software – controlled dynamic voltage and frequency scaling" for different areas on the chip. That means that the frequency of the cores can be slowed down or sped up as needed to provide better energy consumption. Beforehand we want to say that we do not use this feature in our work.

1.3 SCC Chip Components

1.3.1 Overview

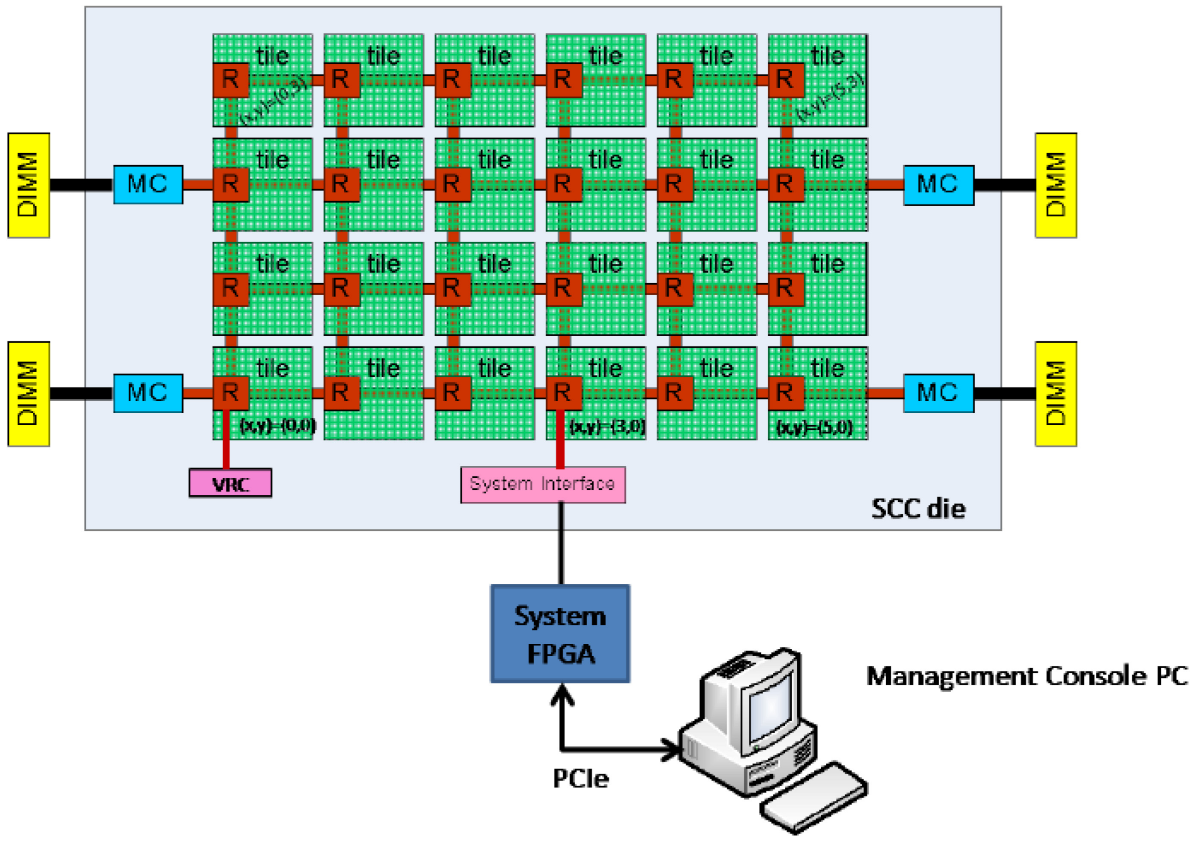


Figure 1, SCC Chip Overview from [4] p. 8

Figure 1 above shows an overview of the SCC chip. The chip mainly consists of 24 tiles that are arranged in a 4x6 matrix. On each tile are two P54c Intel processors with an x86 – Architecture. The tiles also have a routing point “R” that are connected together and form a mesh that each processor can be accessed by the others via the routing points. The chip also has a VRC controller, which takes care of the voltage and frequency regulation, 4 on-die memory controllers (MC) that allow access to external storage and a system interface, which is important to get access to the chip.

1.3.2 Tile Overview

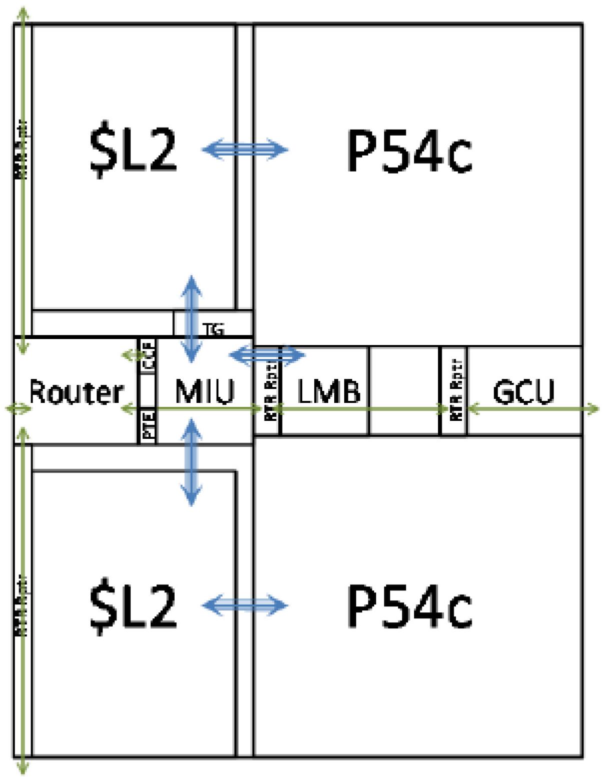


Figure 2, Tile Description from [4] p. 10

A tile has two P54c Intel processors with each of them having a 16KB large L1 data and instruction cache. Both processors are connected to their own private L2 data cache that has 256KB. As shown in figure 2, each tile has a so called Local Memory Buffer (LMB). This buffer is designed for message passing, because every core, even from other tiles, and the system interface have access to the LMB. The LMB is 16KB large and another aspect is that the L2 cache can be skipped to access the LMB. That means that it is not necessary to temporarily save the data in the L2 cache for reading from or writing to the LMB. To handle the incoming and outgoing requests to other tiles or buffers each tile has a Mesh Interface Unit (MIU). The MIU takes care of all network communications. It grants the access to the LMBs and in case of cache misses it handles the communication between the tile and the DDR3 memory. The last part in the figure above is the GCU or Global Clocking Unit. It is responsible for the clock frequency of the mesh and the cores and provides together with the VRC a software managed frequency scaling.

1.3.3 Memory Controller and DDR3 Memory

On the chip are four memory controllers MC (as shown in Figure 1) that provide access to 64GB of DDR3 memory. The DDR3 memory is not directly on the SCC.

1.3.4 System Interface and Management Console PC

The system interface provides access for external PCs to the SCC. Usually SCC users do not access the SCC directly. They login on the external PC, called management PC, write their programs and then launch them on the SCC through the management PC. The management PC can reinitialize the SCC system, boot/reboot the Linux images on the cores, reset the registers and much more on the SCC via the system interface. A picture of the connection from the management PC to the SCC is shown in Figure 1.

1.3.5 VRC

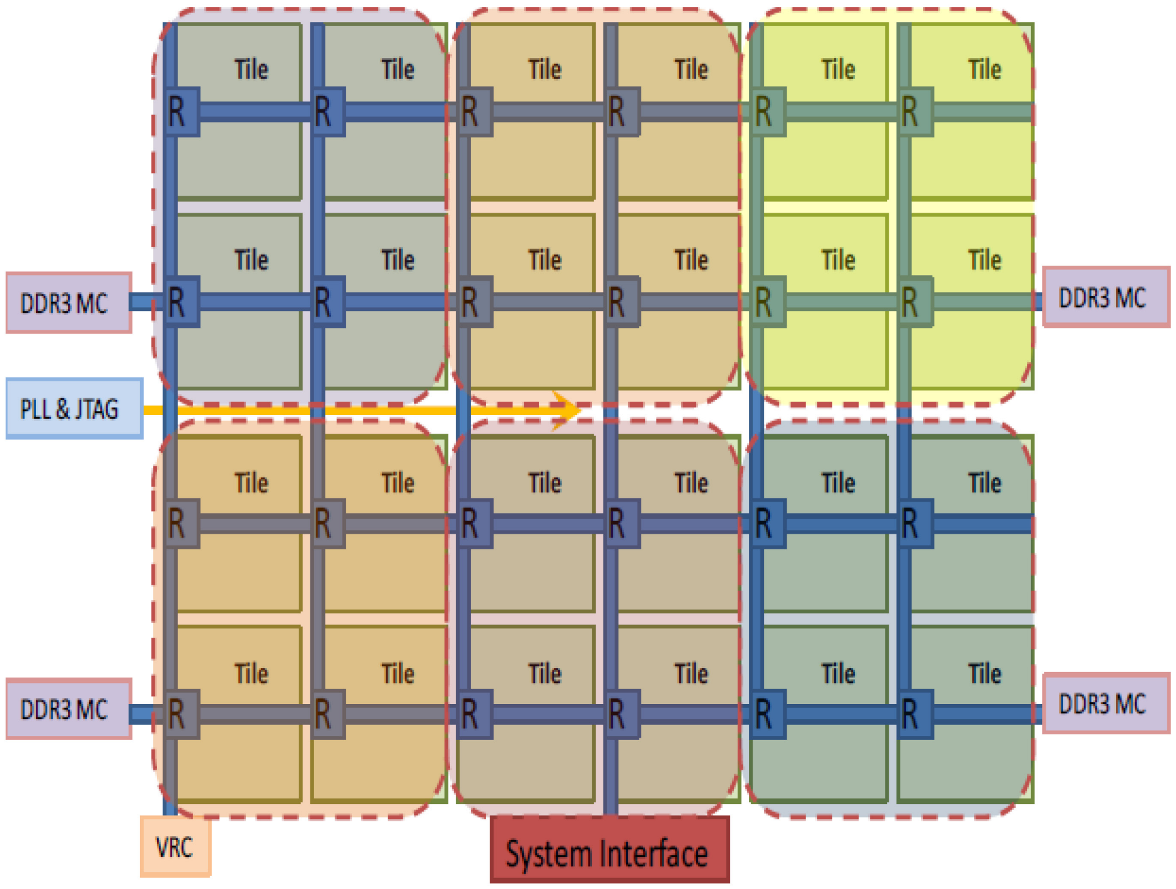


Figure 3, Voltage Regulation Sections from [4] p. 9

The VRC is the voltage regulation controller of the system. Each core or the system interface can change the voltage in any of the areas that are surrounded by the red dotted lines that are shown in Figure 3. In addition the voltage for the mesh can be adjusted too. The most interesting thing about this is that both the power state of the areas and the voltage of the mesh can be fully adjusted by the applications. That means that every core can change the voltage of any of the areas or the whole mesh, as shown in Figure 3 above. When the voltage changes, the cores, the memories and the network can be sped up or slowed down. This opens the possibility of optimal energy consumption, depending on the requirements the running application has. This feature will not be part of our work. We will use a constant voltage that never changes.

2 RCCE

The following chapter is based on [5].

2.1 Introduction in Message Passing

Message passing is a communication paradigm between processes. It is used when data needs to be passed from one process to another one. The data can be seen as a message and the two participating processes are called sender and receiver. Message passing can roughly be divided into blocking and non – blocking communication. These two methods accomplish the same goal that the receiver has the data in the end, but accomplish this goal differently and are used in different ways. For our purposes blocking and non – blocking send and receive functions provide everything we need, therefore we just want to explain these functions and do not go deeper. Our explanations will be based on the Message Passing Interface (MPI) [6, 7 p. 197 - 199]. We want to describe later what exactly this is.

When data shall be passed between processes, both processes (the sending and the receiving) need to provide a buffer. The sender's buffer contains the data that shall be passed and the receiver's buffer shall contain the received data after a successful transaction.

The difference between a blocking and a non – blocking function is the return point of the function. A blocking function returns as soon as the provided buffer can be reused again, without any risk that data may get corrupted. This means for the sender, that the data he wanted to send got successfully transmitted and he can reuse his buffer. The same goes for the receiver. The receiver now has all values from the sender in his buffer and can use it. In a non – blocking send the sending function will return immediately. That means that the call for the send only got initialized but not executed at the moment. The sender now can continue with another part of his program and the data gets sent when there is time for it. The sender of course can check if the buffer got sent yet or not. The same goes for the receiver. A non – blocking receive will return immediately too. It is only an initialization. The receiver may execute some other parts of his program and check in between if the data has been received yet. For both (sender and receiver) exists a wait function that waits until the buffer has been sent or received. This is necessary if the buffer needs to get reused to avoid any data corruption.

For every send a matching receive has to be called. All combinations of blocking and non – blocking functions are possible. A sender can for example send with a blocking send and the receiver can call a non – blocking receive.

After this basic introduction we will now continue with the description of RCCE.

2.2 Introduction in RCCE

The description above was based on MPI [6, 7 p. 197 - 199], which stands for Message Passing Interface. MPI is an international established standard for message passing. It is a very large library for message passing with a huge set of functions. RCCE on the other hand is an MPI like library that is especially designed for the SCC, but with much less functionality. In contrast to MPI, RCCE just provides the important basic functions that are needed for communication between the cores. These will be described in the next chapter.

RCCE provides different interfaces for the programmer, dependent on the required hardware closeness. The simple interface is designed for the majority of developers and abstracts from hardware synchronization flags. For really advanced users it provides a "gory" interface that is much closer to the hardware. The third interface gives us the possibility to control and change the power consumption of different areas, which we described in Figure 3 before. The power consumption is not part of our work. Therefore we will use the simple interface because it provides everything we need. In the following we will only refer to the simple interface.

2.3 RCCEs Functions

The functions below are not all functions that RCCE provides. These are the functions we will use later on in the algorithms. Therefore we want to describe them.

2.3.1 Core Functions

int RCCE_init(int *, char***)
The init function initializes the RCCE engine. It can be called with a parameter list, but we do not use this feature, therefore we will skip further explanations.

int RCCE_finalize(void)
The finalize function is used to shut down the RCCE runtime environment. It has to be the last call to the RCCE library.

int RCCE_num_ues(void)
The num_ues function delivers the number of cores that are used in the program.

int RCCE_ue(void)
The ue function delivers the rank of the calling unit of execution. It can be seen as a core ID. The ID starts from 0 and goes to num_ues – 1. It is the identifier for a core in our algorithms.

int RCCE_wtime(void)
The wtime function is for the time measurement that is necessary for the performance evaluation and returns the wall clock time.

2.3.2 Communication Functions

int RCCE_send(char *, size_t, int)
The send function is a blocking send with the buffer in the first parameter, the length in bytes in the second parameter and the receivers ID in the third parameter. The

function will not return unless the process with the receiver ID calls a matching receive.

int RCCE_recv(char *, size_t, int)
The recv function is the opposite part of the send function. It will return when the whole message has been received and is stored in the buffer.

int RCCE_bcast(char *, int, int, RCCE_COMM)
The bcast function contains a basic broadcast algorithm that is used to distribute data over all participating cores. The RCCE_COMM parameter is the communicator that is used for the algorithm. A communicator contains the cores that take part in the function. It is not necessary that all processes that are used for the full program need to execute the function. As example, if the processes with the IDs from 0 to 7 run the same program, it can be useful that only processes 4 – 7 call the bcast function and the others are not in the communicator. In our case RCCE_COMM is a standard communicator that always contains all processes. We will explain later what a broadcast is.

2.3.3 Synchronization Functions

void RCCE_barrier(RCCE_COMM *)
The barrier function is a synchronization mechanism to ensure that all cores are on the same point in the program. Each core that is in RCCE_COMM has to call this function. The processes can continue with their program when all cores have called the barrier function.

The standard RCCE library only provides blocking functions. For the second algorithm we need non – blocking functions, therefore we use an extension to RCCE, called iRCCE.

2.4 iRCCE

The following chapter is based on [8].

2.4.1 Introduction

As mentioned before RCCE supports only blocking send and receive functions. The problem with that is that in many algorithms blocking send and receive functions will lack in performance. To improve and extend this the iRCCE library was written, which provides asynchronous send and receive functions. In addition to them, iRCCE of course has some functions to check whether a sent buffer has arrived yet or not. Another advantage is that the original send and receive functions match with the new asynchronous functions. For example a sender sends a buffer via a blocking send and the receiver now can call a non-blocking function and they will still match. In the following we want to explain the iRCCE functions that we will use later on in the algorithms.

2.4.2 Core Functions

iRCCE_init();
The init function initializes the iRCCE runtime. It must be called as first iRCCE function, but after the general RCCE_init function. iRCCE does not have a own finalize function. It finalizes with RCCE_finalize.

2.4.3 Communication Functions

int iRCCE_isend(char *, size_t, int, iRCCE_SEND_REQUEST *);
The isend function has the same parameters than the send function from the RCCE library except the request parameter in the end. The request parameter can be seen as a handle to the send request. Since the function returns immediately there is something needed to identify the send request. This is necessary for the test functions to see if the send is completed or not done yet.

int iRCCE_irecv(char *, size_t, int, iRCCE_RECV_REQUEST *);
The irecv function is the same as the isend before, but for the receiver. The request parameter is necessary here too for the same reason. The next function may clarify it better.

int iRCCE_irecv_wait(iRCCE_RECV_REQUEST *);
The irecv_wait function gets called with a receive request. Through the request the function identifies the irecv call and waits until the message from the call gets delivered. An irecv call followed with a wait call has the same semantics as a blocking receive. We did not describe an isend_wait, because we did not use it. For completeness we want to say it is the same as the irecv_wait, but for the sender.

void iRCCE_init_wait_list(iRCCE_WAIT_LIST*);
The init_wait_list function initializes a wait list. A wait list stores all send and receive requests. This is especially useful if there are many send or receive requests that need to get checked for completeness in a loop. The wait list provides many useful functions to handle a larger number requests. We are using only one of them and will describe it later.

void iRCCE_add_to_wait_list(iRCCE_WAIT_LIST*, iRCCE_SEND_REQUEST*, iRCCE_RECV_REQUEST*);
The add_to_wait_list function adds a send or receive request to the wait list. Usually this function gets called immediately after a non – blocking send or receive. It needs to be used if there are multiple send or receives, without any wait between them. The reason is that the requests get stored in a queue and get executed in their incoming order. The first one in gets executed first. The queue guarantees the order and without it could lead into buffer problems.

int iRCCE_wait_all(iRCCE_WAIT_LIST*);
The wait_all function is a very useful functionality of a waitlist. It waits until all requests in the wait list are finished and returns after.

2.5 Memory Organization

RCCE grants access to the on – chip SRAM and to the off – chip DRAM. But there are major differences between the two types of RAMs. The DRAM is mainly handled as private memory, where every core has its own data space. Other cores do not have access to this area. A part of the off – chip DRAM is considered as shared memory among the cores, but it is configurable where this division starts. The shared memory feature in the off – chip memory is not fully implemented yet by Intel. On the other side the on – chip SRAM that is described above in Figure 2 as Local Memory Buffer, is considered as shared memory among the cores. Since every core can access every other LMB the LMBs can be seen as one shared buffer. RCCE divides this buffer logically into 48 parts, for each core one, with 8KB space.

2.6 Caching

The cache behavior for the normal private memory does not differ from other architectures. Private memory gets normally cached through the L1 and L2 caches of the core. The cores and caches on the same tile do not have any kind of algorithms that keep the caches coherent because they are not needed due to separated caches. A special feature is that the shared memory buffers (shared memory in the DRAM and the MPB) can bypass the L2 cache.

2.7 Emulator

RCCE has an emulator that runs under Windows and Linux if OpenMP is supported. The emulator was especially needed before the SCC chip was finished, but today it still can be useful for improving an application, because it delivers a lot of data from a running program. We did not use the emulator for our work.

2.8 Functionalities and Programming Model

2.8.1 Send & Receive, Synchronization and Power Management

RCCEs main functionality is to provide an environment that enables communication, synchronization and power management among the cores. The communication is supported by RCCE through send and receive functions as described before. These functions allow a core to send or receive specific data from another core. All send and receive statements are executed by copying data from the private memory (L2 cache or DRAM) in the MPBs (sending) or from the MPBs in the private memory (receiving). That means that sending and receiving are transactions between the MPBs.

2.8.2 Programming Model and Program Execution

RCCE applications always follow the SPMD (Single Program Multiple Data) scheme [5, 7 p. 93 - 95]. The program executes as one or more instances, where each of them runs exactly the same code. Each instance gets mapped to one core and gets executed until it is finished. It is not possible to swap instances between cores. The instances therefore can be differed by the core ID where they get executed. This works because a core can only run one instance at a given time. It is important that there are no ordering dependencies between the instances, like instance 0 always has to start before instance 1 starts, because the cores start the execution as soon as they receive the instance. As a programmer this needs to be considered that the program works no matter what core begins executing first. If a program does not use all 48 cores that are available on the SCC, it is still not possible to run another one that uses the other cores, because RCCE only allows one parallel program that gets executed at a given time on the chip.

3 PingPong Measurements and Conclusions

3.1 What is it and what is the use of it?

A PingPong test is a small program that measures the time for different sizes of data how long they need to be successfully transferred from one core to another and then returned back afterwards. The idea behind this is to get information about the latency and bandwidth of the given network between the cores. Since our cores are arranged in a matrix, the access times between the cores are expected to differ from each other. Cores that are directly connected together (neighbors) or cores that are on the same tile should not suffer so much from the latency, because they do not have to cross many routing points to gain access to the MPBs of each other. On the other hand if for example core 0, which is in the left bottom corner (as shown in Figure 4 below), wants to send or receive some data from core 47, which is in the right upper corner, needs to cross nine routing points. There are nine routing points, because one is on his own tile and one for each tile the data needs to pass. So the access time should be much longer. This shall be investigated with a PingPong test, because if somehow it is not the case, it would not make sense to implement an algorithm that is especially designed for a given hardware layout, when it cannot take any advantage of it. The exact positions of the cores are shown in Figure 4 below.

37 36	39 38	41 40	43 42	45 44	47 46
25 24	27 26	29 28	31 30	33 32	35 34
13 12	15 14	17 16	19 18	21 20	23 22
1 0	3 2	5 4	7 6	9 8	11 10

Figure 4, Core Arrangement

3.2 Results and Conclusions

Our PingPong benchmarks will be based on Figure 4 above. Core 0 is in each benchmark the core that has the given data. It sends a buffer with a given buffer size to another core and waits until the other core sent the buffer back. We repeated this process 10000 times. Sending the same buffer size over and over seems a bit unnecessary, but the main reason why we did this is to get away from outliners, due to possible network troubles, or other processes that interfere, to get a as clean as possible result. For each of the 10000 runs we measured the time and took the minimum for the performance graphs. The minimum time shows the run with the least interference from any other things that can influence the time. The following code snippets show the code that we used for the measurements.

```
35  for(i = 0, size = START_SIZE; size <= MAX_SIZE; i++, size += STEP_SIZE)
36  {
37
38    sizes[i] = size;
39
40    buffer = (char*)malloc(size * sizeof(char));
41
42    for(k = 0; k < size; k++)
43      dummy += buffer[k];
44
45    if(ME == 0)
46    {
47      for(k = 0; k < size; k++)
48        buffer[k] = k % 127;
49      printf("\nSIZE: %d\n", size);
50    }
51
```

The i loop in line 35 represents the buffer sizes. i counts the rounds and $size$ goes from a given start size with a step size to the maximum size. We will describe the used sizes later in the next chapter. The $buffer$ variable is the buffer that we will use for the performance measurements. In line 40 we allocate a buffer with the actual size from the $size$ variable. At the end of the loop the buffer gets freed that it can be reallocated in the next run. We do not show this part. The lines 42 – 43 shall ensure that the buffer is in the nearest cache. That is needed to eventually copy the buffer to the L2 cache from the off – chip DRAM memory if it got allocated there. In line 48 the process with the ID 0 sets the buffer values to later check if the buffer was transferred correctly. These lines are necessary for the different sizes that we will test. The next snippet will show the actual PingPong code.

```
52      for(j = 0; j < 5; j++)
53      {
54        total_time[j][i] = 0.0;
55
56        partner = processors[j];
57
58        RCCE_barrier(&RCCE_COMM_WORLD);
59
60        for(x = 0; x < ROUNDS; x++)
61        {
62          if(ME == 0)
63          {
64            RCCE_send(buffer, size, partner);
65            RCCE_recv(buffer, size, partner);
66          }
67          else
68          {
69            if(ME == partner)
70            {
71              RCCE_recv(buffer, size, 0);
72              RCCE_send(buffer, size, 0);
73            }
74          }
75          if(x == 0)
76            timer = RCCE_wtime();
77        }
78
79        timer = RCCE_wtime() - timer;
80        total_time[j][i] = timer;
```

The code lines above show the kernel of the PingPong test. Before we want to start in line 56 with our explanations, we want to say that this code snippet will be done for each buffer size. The $processors$ array stores the core IDs from the cores that we want to use in our test. The IDs we chose are 1, 2, 10, 37 and 47. Why we chose this cores will be described in the next chapter. The j loop in line 52 goes through the $processors$ array to perform the PingPong Test with each core. This is the reason why it only goes from 0 to 5. The $partner$ variable in line 56 gets the core ID for this round. Before starting with the benchmark we perform a barrier for the synchronization between the cores, which is shown in line 58. The PingPong benchmark is now done in the lines 60 – 77. The loop in line 60 performs the benchmark 10000 times. The time gets measured after the first run, when all buffers have the data once to prevent any kind of possible caching problems. After the benchmark is finished the time gets stopped and is written in the $total\_time$ array for the core in the jth position and the ith data size. The following figure shows the results of the test.

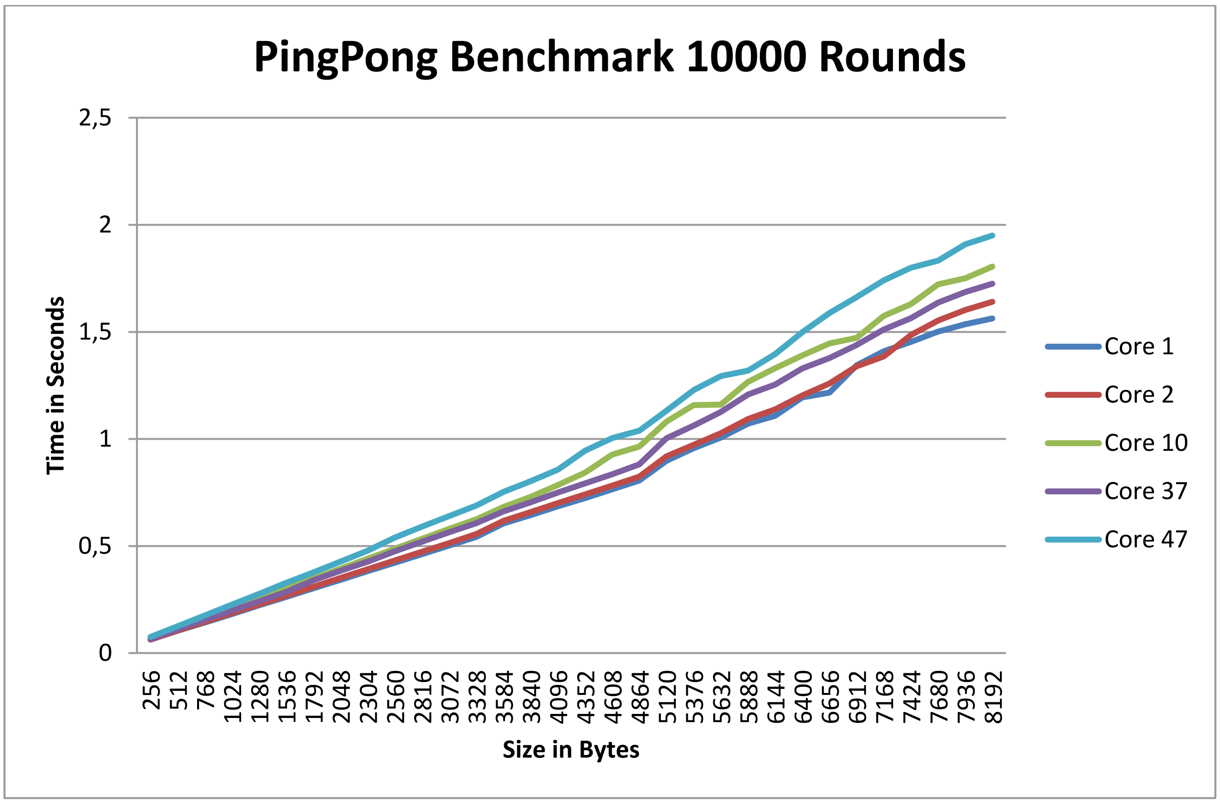


Figure 5, PingPong Test Results

The test was made for all different cores with the same sizes. The buffer size began at 256 byte and ended at 8 kilo byte with a step size of 256 bytes. We stopped at 8 KB, because this is the maximum size that can be stored in the Message Passing Buffer. All larger data sizes have to be at least transferred into a cache, which costs time and influents the results. The cores were chosen by their different distances in the matrix. Core 1 is on the same tile as our starting core 0. That's why it should need the least time. Core 2 is on the neighbor tile of core 0 and core 1. That means that it is only two routing points away (one on the tile of core 0 and core 1 and one on its own tile). Core 10 and core 37 represent the end of the same row or column and core 47 is on the furthermost tile. For clearer results we sped up the core frequency from 533 to 800 MHz and the buffer frequency from 800 to 1066 MHz. The results show that the needed time scales with the distance between the cores. Core 47 needs way more time than core 1 or core 2.

4 Broadcast Algorithms

4.1 Introduction

A broadcast [7 p. 119-120] algorithm distributes a buffer from the root process (the process that has the data) to all other processes in the network that take part in the operation. Figure 6 below will show the process.

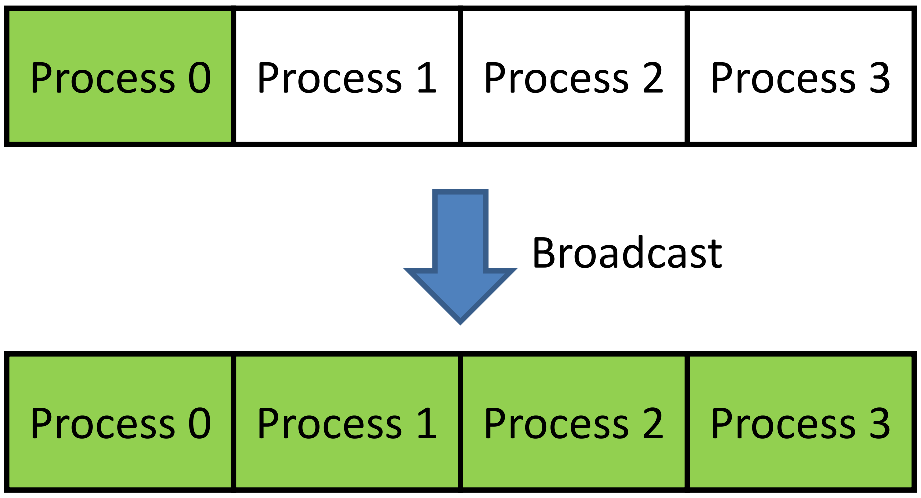


Figure 6, Broadcast in General

We assume that process 0 is our root process that contains the data that shall be distributed over all other cores. In the end all cores that participated in the process shall have the same data.

The interesting question is how to accomplish this goal?

The easiest approach would be that process 0 sends the data to process 1, then to process 2 and at last to process 3. The problem with this is that it would take three rounds to finish and process 3 has to wait all three rounds before it can start working with the data. For only 4 processes this does not seem to be a big problem, but in modern multiprocessor architectures with sometimes 100000 processors it would be crucial, when all processors have to wait until all before have been delivered. The problem with this approach is that not all processes are used when they already received the data. Only process 0 passes the data to other processes. After sending the data to process 1, two processes have the data and can send them to the next processes. If it is done correctly the number of processes each round that can send the data to the next doubles. Imagine the example before, but with 8 cores. With the first approach it would take seven rounds to finish the broadcast, with the second only three. With the condition that a process can perform only one send operation in a round. The more processors the larger gets the gap, because we changed the order of this algorithm. The first one needed $O(n)$, where n is the number of processors. The duration of the algorithm scales linear to n. In the second approach the order changes to $\mathrm{O}\lceil\log_2 n\rceil$, which will result in a huge performance improvement. This order is the theoretic optimal solution for algorithms that accomplish the same problem, when the whole data is treated as one part. It is important to say that the above described formulas do not include any network startup times or bandwidth considerations.

This leads into the question why do so many different algorithms exist, if there seems to be an optimal solution in the number of rounds?

The problem with this solution is that it is only theoretical. In reality there are not always 2^n processors. Another problem is that it is not optimal for each data size to be sent with only one sending operation. It is very often the case that splitting up the data in multiple parts is way faster than sending the whole in one part. To fulfill these goals you need an algorithm that can deal with an odd number of processors, multiple data chunks and still should provide a near optimal performance. We will now start with the description of the two broadcast algorithms.

5 General Approach

5.1 Introduction

Our first algorithm that we want to investigate and implement is the algorithm from Bin Jia [9]. The algorithm is designed for a fully connected homogenous network, where each processor can communicate with any other processor. How exactly the underlying hardware is designed does not matter. The algorithm is considered as round based. In each round a processor can send a message to a processor and receive a different message from another processor. For each round it is exactly determined for each processor which message shall be sent and to which process and vice versa for receiving. A round is always a complete action and the next round will not start before the previous has been finished. The algorithm provides the optimal number of rounds to distribute the whole data over the network. This number can be calculated as followed. It depends on the number of parts and the number of processors that were used. The parts represent the amount of chunks in which the data will be split up. It takes $log(n)$ rounds to finish the broadcast if we always send the whole data, as described before. When we use multiple parts we need to adjust this a little bit, because it takes the number of parts rounds to pass every piece from process 0 to the next one. Therefore all in all this results in $parts + log(n)\ rounds - 1$. This is the mathematical lower bound for algorithms like this. Our algorithm differs if the number of processors is a power of two or not. If it is a power of two, the algorithm is simpler and therefore called the basic algorithm. The arbitrary version for any amount of processors builds on the basic algorithm and extends it. Therefore we will start our explanations with the basic algorithm.

5.2 Power of 2 Processors

A broadcast for a power of 2 number of processors is a well-known problem as investigated in [10, 11]. Both of these algorithms are theoretical optimal, but as described by Bin Jia in [9], they are not very practicable to implement. We want now describe the algorithm from [9].

In every round a process has a partner process. The partners will change from round to round. A process interacts only with his partner process. All possible send and receive operations are done between it and its partner. If for example process 2 has calculated process 6 as its partner, then process 6 also gains process 2 as partner as result from the calculation. It is not possible that the calculations of process 6 would lead into another partner process. The partner process is defined by the bit pattern of the ID of a process. For example process 2 has a binary bit pattern that looks like this (0010). In the first round the process flips the first bit from the right. The result will be (0011). That means that process 3 is the partner process in the first round. To prove the correctness, when process 3 does this it receives process 2 with (0010) as partner. The partner determination needs to be done every round. The following formula shows the partner determination for a given round. All formulas that we use are from [9].

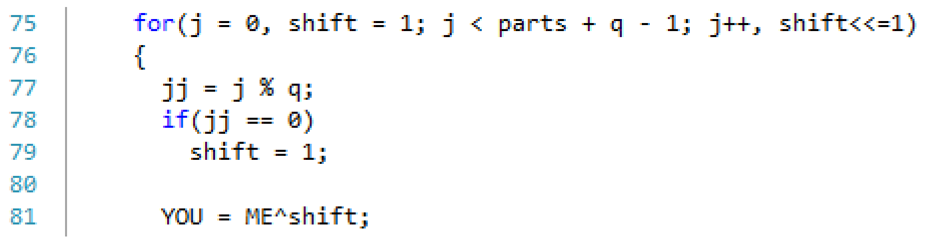
$$Partner(i,j) = \left(i_{q-1} \dots \bar{i}_{j'} \dots i_0\right)_2$$

The variable i stands for the process ID. j is the counter for the rounds that shows the current round. The expression in the brackets represents the binary bit pattern of the process i. The expression $\bar{i}_{j'}$ in the middle represents the flip of the j' bit in the bit pattern of the process. The q variable stands for the power of 2 to gain n (number of processors), or simpler $q = \lfloor \log(\mathrm{n}) \rfloor$. There is one variable left and this is j'. j' is depended on j, because $j' = j \bmod q$. j' gives us the index of the bit that shall be changed. Summing up the explanations and the formula, the partner process of a process i in the round j is determined by flipping the j' bit from the right in the binary representation of the process. We only look at the first $q-1$ bits, because every possible combination of 0s and 1s is assigned to a process ID, because we have 2^qprocessors. For a better understanding we want to relate to the following table that shows the partner process for every process.

q	j	j'	i = 0	Partner	i = 1	Partner	i = 2	Partner	i = 3	Partner
2	0	0	00	01	01	00	10	11	11	10
2	1	1	00	10	01	11	10	00	11	01

We have 4 processes. They are represented by their value of i. The processors are numbered from 0 to 3. The column j shows the rounds. q is a constant value, because the number of participating processes does not change. In the first round every process flips the rightmost bit in their binary representation. The resulting process pairs are (0:1, 2:3). This determination needs to be done in every round from every process.

At this point we want to add some pieces of our program code for further comprehension.



```
75      for(j = 0, shift = 1; j < parts + q - 1; j++, shift<<=1)
76      {
77        jj = j % q;
78        if(jj == 0)
79          shift = 1;
80
81        YOU = ME^shift;
```

Our counter j starts from 0 and runs until $parts + q - 1$ as described before. This is the theoretically optimal number of rounds. The variable jj in line 77 is the same as j'. The check in line 78 is necessary, because we only use the last $q - 1$ bits of the processor ID. The ID is shown in ME and is a normal 4 byte integer as all of these variables. We use the variable $shift$ for actually flipping the bit. The $shift$ variable does a left shift at the beginning of each round, except the first one. Therefore it starts with a 1 in the last bit and the other bits are sets to 0. After each round the 1 changes its position and moves a bit left in the representation. In the decimal system it would be a multiplication with 2. The $shift$ variable always consists of zeroes except the one bit that shall be flipped, this is always a 1. The flip is then done by an exclusive or in line 81. It does not matter if the bit, where $shift$ has a 1, is a 0 or a 1, it always flips. This process needs to be done every round by every process. After determining the partner process a process needs to calculate which parts shall be sent and received.

The calculation of the parts that shall be sent and received is a little bit more complicated. We will again show the formula how to calculate it and describe it afterwards.

$$s(i,j) = j - q + \left(1 - i_{j'}\right) * Dis_i[j']$$
$$t(i,j) = j - q + i_{j'} * Dis_i[j']$$

The s variable stands for send and the t variable for receive. They represent the part that shall be sent or received. If s or t is less than zero, the operation gets skipped and nothing will be sent or received. It is also possible that s or t are greater than the number of parts, then they get set to the last part. The $Dis\ Array$ is an q – sized array that each process creates, based on its own binary bit pattern. The array needs to be created only once at the beginning. It then contains all necessary data for the algorithm. The formulas can be split up in different terms. The term $j - q$ acts like a barrier in the first rounds, that prevents processes from sending and receiving, because j starts at 0 and it will add up a negative term to the end result until it reaches q. This causes many send and receive operations to get skipped in the beginning, because s or t is less than 0, which is necessary. We will show an example later, for a better understanding. The second term is a distance calculation that is necessary to determine the part that can be sent to or received from the partner. The terms $(1 - i_{j'})$ and $i_{j'}$ in the distance calculation bring in the bit swap in the determination of the partner process. Therefore it is ensured that $s(i, j)$ of a process is the same as $t(i, j)$ of its partner process. Each process needs to build up its own $Dis\ Array$ based on its binary representation. The process starts with the last element of the array. The same goes for the bit in the binary representation. He then counts how many bits are needed to find the first nonzero bit to the left and writes the number in the $Dis\ Array$. This process gets repeated for every element in the array. A wraparound is allowed. When he reaches the left end of the binary bit pattern and no nonzero bit has been found he continues with the most right bit. Process 0 has an exception from this schema, because it only has zeroes in the binary representation, therefore every element in the array gets the value of q. For an easier understanding we want to show an example of the $Dis\ Array$ that is also from [9].

The example is for the processor = i = 406 and the number of processors = n = 512

j	8	7	6	5	4	3	2	1	0
i_j	1	1	0	0	1	0	1	1	0
$Dis_i[j]$	2	1	1	2	3	1	2	1	1

j in the table above indicates the position in the array. i_j represents the bit at the index of j of the process i that is 406 in the example. To determine the $Dis\ Array$ one has to start at position $j = 0$. Then count how many bits to the left it takes to find a 1. For index 0, it needs one bit. The count gets stored in the $Array$. The process repeats with index 1 and so on. At the end the $Dis\ Array$ looks like above. After the theoretical explanation we want to show our implementation of this part.

```
    dis = (int*)malloc(q * sizeof(int));

    if(ME == 0)
    {
      for(i = 0; i < q; i++)
        dis[i] = q;
    }
    else
    {
      for(head = 0, shift = 1; head < q; head++, shift<<=1)
      {
        if((ME & shift) == shift)
        {
          for(cur = tail; cur < head; cur++)
            dis[cur] = head - cur;

          tail = head;
          if(rightmost == -1)
            rightmost = head;
        }
      }
      for(cur = tail; cur < q; cur++)
        dis[cur] = q - cur + rightmost;
    }
```

The algorithm is straight forward implemented from the pseudo code that is used in [9]. The $Dis\ Array$ is named dis in the code above. Every processor calculates it before the actual algorithm starts, because it only has to be calculated once. $tail$ gets initialized with 0 and $rightmost$ with -1. Before we will show how we implemented the whole basic algorithm, we want to show an example how the whole basic algorithm works.

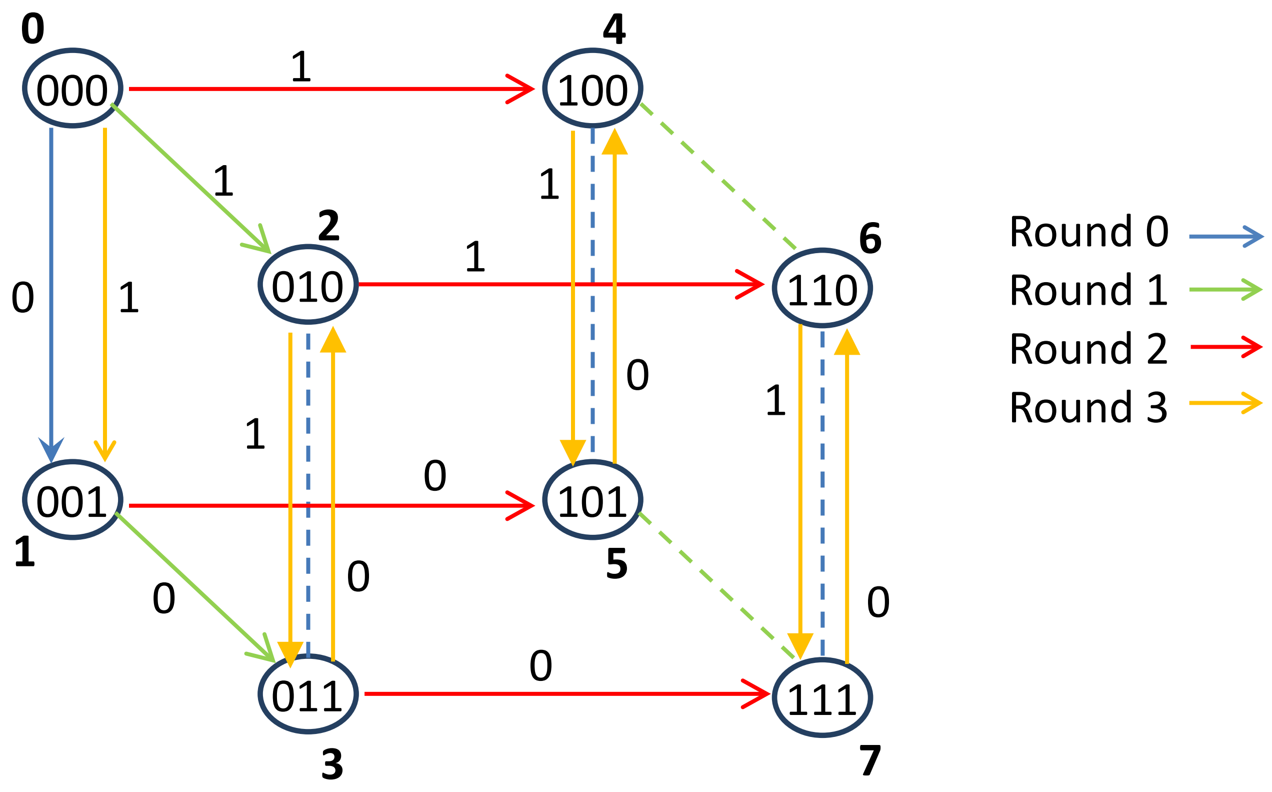


Figure 7, Explanation of the Basic Schema

The figure above is an example how the algorithm works. It is used with 8 processors in total and two message parts. The different line colors as shown in the legend above indicate the different rounds. In general, all arrows show data movement and all dashed lines show the connection to the partner process, but there is no data movement this round. The first round, in our case with round index 0, is marked with blue arrows. Therefore each process flips the last bit to determine its partner process. Process 0 has process 1 as partner, process 2 has process 3 and so on. Data transfer only happens between process 0 and process 1, as we can see through the arrows. All others do not have valid send and receive indexes. In round two, which is shown through the green line, the bit in the middle flips to calculate the partner. In this round process 0 and process 1 can send their data to the next processes. Process 0 sends the second part, in our case part number one and not the first one to distribute as many different parts as possible to other processes. The third round is marked through the red line and every process flips their first bit. Now we have four sending processes. The last round that is marked by the yellow arrows is the most interesting one. It shows that the processes have the same partners as in the first round. That means that a process always has the same partners repetitive throughout the algorithm, no matter how many different parts need to be sent. As we said in the beginning, a process can send and receive a part in a single round. This is shown in the last round by the yellow arrows. We will call the algorithm Cube algorithm, because it looks like this in the example. In the following we want to show the implementation.

```
75      for(j = 0, shift = 1; j < parts + q - 1; j++, shift <<= 1)
76      {
77        jj = j % q;
78        if(jj == 0)
79          shift = 1;
80
81        YOU = ME ^ shift;
82
83        bit = ((ME & shift) == shift) ? 1 : 0;
84
85        s = j - q + (1 - bit) * dis[jj];
86        t = j - q + bit * dis[jj];
87
88        if(s > parts - 1)
89          s = parts - 1;
90
91        if(t > parts - 1)
92          t = parts - 1;
93
94        if(ME < YOU)
95        {
96          if(s >= 0)
97            if(s == parts - 1)
98              RCCE_send(buffer + s * part_size, last_size, YOU);
99            else
100             RCCE_send(buffer + s * part_size, part_size, YOU);
101
102         if(t >= 0 && ME > 0)
103           if(t == parts - 1)
104             RCCE_recv(buffer + t * part_size, last_size, YOU);
105           else
106             RCCE_recv(buffer + t * part_size, part_size, YOU);
107       }
108       else
109       {
110         if(t >= 0)
111           if(t == parts - 1)
112             RCCE_recv(buffer + t * part_size, last_size, YOU);
113           else
114             RCCE_recv(buffer + t * part_size, part_size, YOU);
115
116         if(s >= 0 && YOU > 0)
117           if(s == parts - 1)
118             RCCE_send(buffer + s * part_size, last_size, YOU);
119           else
120             RCCE_send(buffer + s * part_size, part_size, YOU);
121       }
122     }
```

As explained before every process calculates its partner. This is done from 77 to 81. After this the parts that shall be sent and received get calculated. This is done from 83 to 92. When the calculations are finished the actual sending and receiving can begin. There is one important thing we need to take care of and this is to prevent deadlocks. Therefore we made the if – statement in line 94. Without this statement partners would maybe both send or receive at the same time and cause a deadlock, because they would both keep waiting. The if – statement prevents this by ordering the operations due to the processor ID. Before a send or receive function gets executed it gets checked if it is valid. If s or t is below zero the operation gets skipped. We will now follow up with the arbitrary version of the Cube algorithm.

5.3 Arbitrary Number of Processors

The arbitrary algorithm builds on the algorithm described before. The round based idea is still the same and the complexity as well. The algorithm needs the exact same number of rounds as the basic algorithm before. The main change in the new version is that the processes are combined together in so called units. A unit consists of either one or two processes. This results in a total of 2^qunits. q is the same as before: $q = \lfloor \log(\mathrm{n}) \rfloor$. For example if there are 7 processors, q will be two. This results into one unit with one processor and three units with two processors, summed up into four units. The communication in a round is from now on from unit to unit. The unit a process belongs to is for the whole algorithm and never changes. Every round each process has to determine who in the unit is the process that receives the message for the unit and who is the one who sends the message for the unit. They calculate too if they need to pass a message to the partner in the unit or not. This is necessary because if they would not do this, one process would miss the data. In the end the algorithm takes care if every message has been sent to each unit, not to each process, therefore they need to pass the messages to their partner in the same unit.

Each process starts by defining his partner. We want to show how this is done through our source code.

```c
int calc_co(int p, int powerofq, int n)
{
  int co;

  if(p == 0)
    co = 0;
  else
  {
    if(p <= (n - powerofq))
      co = powerofq - 1 + p;
    else
    {
      if(p < powerofq)
        co = p;
      else
        co = p - powerofq + 1;
    }
  }
  return co;
}
```

The partner process of the same unit is named co that stands for cooperation process. In our source code, p represents the process ID. A process that's ID is lower than 2^q will be paired up with a process that's ID is higher than 2^q or equal 2^q. Process 0 is an exception, because it is never paired up. The variable $powerofq$ is the result of the calculation 2^q and n stands for the number of processes.

To represent the unit we will introduce a new variable called rep. If the unit has two processes rep will be the one with the lower ID.

```c
    rep = (ME < powerofq) ? ME : co_i;
```

If the unit consists of only one processor then rep gets his ID. In [9] they represent the following formula to define which process is the sending (out) process of the unit.

$$Out(i,j+1) = \left(1 - Rep(i)_{j'}\right) * Out(i,j) + Rep(i)_{j'} * In(i,j)$$

As we can see this is an iterative formula, because the result builds on the previous one. For the first round (j=0) we have to determine the starting values. Therefore the in process gets defined with rep, that means the smaller one, and the out processor is $\mathrm{co(rep)}$, which is his partner. The role switching is of course only necessary if two processors are in the unit. So for the first round the processor with the smaller ID is the one who receives data and his partner is the one who sends data. According to the formula above the roles can change every round. The role switches depend on the binary representation of rep. With the formula above the role switching in the own unit can be handled, but there is also a problem with this solution. In every round, each unit has a partner unit. Usually each unit sends a message to the partner unit and receives a message from the partner unit. In our unit we know exactly who the sender is and who the receiver is, because we use our formula, but we do not know it for the partner unit. We need to know it because a data exchange can only happen between two processes. Therefore we need to know which of the processes in our partner unit is in this round the sender and who is the receiver. We could calculate it with the formula above, but the problem is that the calculation is iterative. That means we need to calculate it for all previous rounds and in the next round we get a new partner unit and have to recalculate it again. This does not seem to be an effective idea. We know that the role switches depend on the nonzero bits of the binary representation of rep. To solve the problem in [9] they introduce a switch array that stores the number of the 1s in the binary bit pattern of rep between bit 0 and bit j. At this point we want to show an example of the switch array.

The example is for the representator of the unit = rep = 406 and the number of processors = n = 512

j	8	7	6	5	4	3	2	1	0
rep_j	1	1	0	0	1	0	1	1	0
$Switch_i[j]$	5	4	3	3	3	2	2	1	0

Therefore the number of role switches before a given round j can be determined as followed.

$$u(i,j) = Switch_i[q-1] * \lfloor (j-1)/q \rfloor + Switch_i[(j-1) mod\ q]$$

This leads to a new calculation of the role determination.

$$Out(i,j) = (1 - u(i,j)') * Co\big(Rep(i)\big) + u(i,j)' * Rep(i)$$

In the formula above u(i,j)' means u(i,j) mod 2.

We know that the bits in the binary representation of our partner unit only differ in a single bit. This knowing can be used to speed up the calculations for the partner units. The formula for calculating the out process of the partner unit is as follows.

$$\begin{aligned} &Out(Partner(Rep(i),j),j) \\ &\qquad = \big(1 - v(i,j)\big) * Co\big(Partner(Rep(i),j)\big) + v(i,j) \\ &\qquad * Partner(Rep(i),j) \end{aligned}$$

The term $v(i,j)$ stands for $(u(i,j) \ + \ j/q) \ mod \ 2$, which brings in the consideration of the one bit that changed for the partner unit. We want to show at this point our program code for further explanation and better understanding.

The program starts by calculating all static parts. We consider parts as static parts, when they only have to be calculated once and then keep their values and never change. The first thing each process has to do is calculating its partner process and the representative for his group. This is done in the following lines of code.

```
    co_i = calc_co(ME, powerofq, n);

    rep = (ME < powerofq) ? ME : co_i;
```

The $calc\_co$ function in the code snippet above is the same function as described before and calculates the partner for a given process. The result is saved in co_i. The rep variable is the representative of the group and gets the smaller ID if it is a group with two processors in it. After these two small things we have the $Dis\ Array$ and the $Switch\ Array$ left for our static calculations. The $Dis\ Array$ is the same as described before in the basic algorithm, therefore we will skip this part here. The $Switch\ Array$ is calculated as followed.

```
160     counter = 0;
161     for(i = 0, shift = 1; i < q; i++, shift <<= 1)
162     {
163       if((rep & shift) == shift)
164         counter++;
165
166       switcharr[i] = counter;
167     }
```

The $Switch\ Array$ once again is important for the determination of the role switches. In our code it is named $switcharr$. It stores the number of 1s in the binary bit pattern of rep from 0 to in our case named i, in the code above. i runs from 0 to $q - 1$, because we do not have more groups. After calculating all this things the iterative part of the algorithm starts.

```
169     for(j = 0, shift = 1; j < parts + q - 1; j++, shift <<= 1)
170     {
171       jj = j % q;
172       if(jj == 0)
173         shift = 1;
174
175       u = (j > 0) ? switcharr[q-1] * ((j - 1) / q) + switcharr[(j - 1) % q] : 0;
176       co = calc_co(rep, powerofq, n);
177       out = (1 - (u % 2)) * co + (u % 2) * rep;
178       in = calc_co(out, powerofq, n);
179       partner = rep ^ shift;
180       v = (u + j / q) % 2;
181       co = calc_co(partner, powerofq, n);
182       out_partner = (1 - v) * co + v * partner;
183       in_partner = calc_co(out_partner, powerofq, n);
184
185       bit = ((rep & shift) == shift) ? 1 : 0;
186       s = j - q + (1 - bit) * dis[jj];
187       t = j - q + bit * dis[jj];
188
189       if(s > parts - 1)
190         s = parts - 1;
191
192       if(t > parts - 1)
193         t = parts - 1;
```

The code snippet above shows all calculations that have to be done for each round by each process, before they can start sending and receiving data. Important is that all these calculations are static calculations with a complexity of O(1). We want to start with our loop. As one can see in line 169 the loop goes from 0 to $parts + q - 2$, which is exactly the amount of runs that we said before is the theoretically optimal number of rounds. The variable $shift$ is only a helper that doubles each round until the actual round j reaches q, then it gets set back to 1.The first thing each process has to do is now to determine if it is the sender, in the code above called out, or the receiver, in the code above called in, in his group. If a group consists only of a single process, this process is the sender and the receiver. First we need to calculate the switches in the group that have occurred before a given round j. This is done in line 175. The variable u represents this. As we can see the code looks like the formulas before. A special case is the first round, because there was no round before and this would lead into an undefined state by the access of the $Switch\ Array$. In line 176 each group member calculates a defined cooperation partner of the group representative rep. rep and co shall have the same values for both group members, because this will be used afterwards. And both processes in a group shall receive the same result of who is the sender and who the receiver. For a group with a single process, the function call is not necessary. Now we can determine the in and the out process. The out process gets calculated in line 177 with the formula from above. The result gets saved in out. As we can see the variable co is needed for the calculations to ensure that both processes gain the same result. Calculating the in process is of course way easier, because the in process has to be the cooperation partner of the out process. Therefore we can calculate it due to our cooperation partner function, which is done in line 178. Our communication partner gets calculated like before in the basic algorithm. This is done in line 179. The only difference is that it is a group this time. Therefore the in and out processes of our partner group have to be determined too. They get calculated from the formulas above in the lines 180 to 183. The variables $in\_partner$ and $out\_partner$ represent the in and out process of the partner group. After getting all information about the different roles, the processes have to calculate which parts shall be sent and received. This is the same as before in the basic algorithm, with one little difference. The change is in line 185. Every process fits in the representative for the group instead of his own ID. The two if statements are necessary, because s and t could be more than the number of parts and in this case they shall be set to the last part. These calculations finish the calculations that need to be done for every round. The processes have now all information to begin with the send and receive operations. When starting the send and receive operations the processes have to differ if their group consists of one or two members. If it consists of only one member they have to be aware not to cause deadlocks. Deadlocks can happen if it is not well defined when to receive and when to send first. The following code snippet shows the send and receive operations and the deadlock avoidance.

```
195       if(ME == co_i)
196       {
197         if(ME < in_partner)
198         {
199           if(s >= 0)
200             if(s == parts - 1)
201               RCCE_send(buffer + s * part_size, last_size, in_partner);
202             else
203               RCCE_send(buffer + s * part_size, part_size, in_partner);
204
205           if(t >= 0)
206             if(t == parts - 1)
207               RCCE_recv(buffer + t * part_size, last_size, out_partner);
208             else
209               RCCE_recv(buffer + t * part_size, part_size, out_partner);
210         }
211         else
212         {
213           if(t >= 0)
214             if(t == parts - 1)
215               RCCE_recv(buffer + t * part_size, last_size, out_partner);
216             else
217               RCCE_recv(buffer + t * part_size, part_size, out_partner);
218
219           if(s >= 0)
220             if(s == parts - 1)
221               RCCE_send(buffer + s * part_size, last_size, in_partner);
222             else
223               RCCE_send(buffer + s * part_size, part_size, in_partner);
224         }
225       }
```

In line 195 we can see the check if a process is his partner process. If this is the case it is clear that the group consists of only one process. A deadlock can happen if two groups with only one process each try to communicate. If they both send or receive at the same time it will result in a deadlock, because both processes will be waiting for the other one. The easiest way to fix this problem is to define a fixed order. This is done in line 197. The enquiry gives the process with the lower ID the permission to send first, as we can see in the following lines. On the other side the process with the higher ID has to receive first. Through this simple schema we have a defined order and successfully prevented deadlocks. The sending and receiving on its own is pretty much the same than in the basic algorithm.
The following code snippet shows the send and receive operations for normal groups, with 2 processes in it.

```
228      if(ME == out)
229      {
230        if(s >= 0)
231          if(s == parts - 1)
232            RCCE_send(buffer + s * part_size, last_size, in_partner);
233          else
234            RCCE_send(buffer + s * part_size, part_size, in_partner);
235
236        if(j - q - 1 >= 0)
237          if(j - q - 1 == parts - 1)
238            RCCE_recv(buffer + (j - q - 1) * part_size, last_size, in);
239          else
240            RCCE_recv(buffer + (j - q - 1) * part_size, part_size, in);
241      }
242      else
243      {
244        if(t >= 0)
245          if(t == parts - 1)
246            RCCE_recv(buffer + t * part_size, last_size, out_partner);
247          else
248            RCCE_recv(buffer + t * part_size, part_size, out_partner);
249
250        if(j - q - 1 >= 0)
251          if(j - q - 1 == parts - 1)
252            RCCE_send(buffer + (j - q - 1) * part_size, last_size, out);
253          else
254            RCCE_send(buffer + (j - q - 1) * part_size, part_size, out);
255      }
256    }
```

The processes have to determine whether they are the sender or the receiver this round. This is done in line 228. The sender then sends his part to the in process of the partner group, which is done in the lines 232 and 234. The check in line 236 is to determine if a part needs to be received from the in process of the same group, to keep the data consistent for both processes. If the process is the in process this round, it receives a part from the out process of the partner group, which is shown in the lines 246 and 248. Later some parts may be passed to the out process of the own group.

At the end of the loop, there are 2 parts left in each group that need to be passed to the group partner if it is a group of two processes. This is shown in the following.

```
    if(ME != co_i)
    {
      u = switcharr[q - 1] * ((j - 1) / q) + switcharr[(j - 1) % q];
      co = calc_co(rep, powerofq, n);
      out = (1 - (u % 2)) * co + (u % 2) * rep;
      in = calc_co(out, powerofq, n);

      if(ME == out)
      {
        RCCE_recv(buffer + (parts - 2) * part_size, part_size, in);
        RCCE_send(buffer + (parts - 1) * part_size, last_size, in);
      }
      else
      {
        RCCE_send(buffer + (parts - 2) * part_size, part_size, out);
        RCCE_recv(buffer + (parts - 1) * part_size, last_size, out);
      }
    }
```

The groups have to recalculate their roles and then pass the missing parts to their partner. This is not done in the loop because it is only group intern and does not affect groups with only one process in it.

5.4 Proof of the Algorithm

We did not proof the correctness of the algorithm. The correctness part can be found in [9]. We accepted the given formulas and program flows.

5.5 Getting out the best Performance

As mentioned before the algorithm is theoretical optimal for a given number of parts, but the actual runtime on a given hardware varies when we change the number of parts that we want to use in the algorithm. We want to explain this behavior with the following example.

Imagine that we have one megabyte of data that we want to distribute among the processors. The question is shall we consider it as one part and distribute it over the network, or shall we split it up in a thousand parts. The advantage of one part obviously is that we do not need many send operations to finish the algorithm and therefore do not suffer that much from the latencies of the network. On the other hand splitting up the parts results in better pipelining, because we send smaller parts and therefore all processes can begin their work earlier, but we have to deal more with network latencies. Additionally to these considerations we need to adjust the amount of parts to the cache sizes of the processors. As we can see it is difficult to say what the best number of parts for the algorithm is.

To answer this question we have to look closely at the hardware setup that we are using. Our processors on the SCC – Chip have an 8 KB message passing buffer each. That means one send operation can only send 8 KB in one operation. Therefore it makes no sense to choose a larger part size than 8 KB, because our algorithm basically sends one piece a round. It could be that smaller sizes than 8 KB could be better. This depends on the latency and the bandwidth of the network. Therefore we have tested our algorithm with different sizes. Before we said that it makes no sense to use larger sizes than 8 KB, but we tested some of them too just to validate this assumption. Sometimes the results do not show the expectations. The following table shows all results.

	50.000 Bytes		190.000 Bytes		1.900.000 Bytes	
Part Size	Min Time	Avg Time	Min Time	Avg Time	Min Time	Avg Time
524288	x	x	x	x	0,827701	0,829792
262144	x	x	x	x	0,701350	0,702657
131072	x	x	0,035007	0,038718	0,651371	0,652171
65536	x	x	0,026863	0,027705	0,638883	0,639950
32768	0,006345	0,006867	0,024891	0,025497	0,618031	0,619026
16384	0,004690	0,004715	0,025908	0,026839	0,598216	0,598796
8192	0,003882	0,003895	0,024354	0,025255	0,583514	0,583881
4096	0,003405	0,003445	0,025761	0,026348	0,572945	0,573236
2048	0,003074	0,003178	0,026584	0,027242	0,571021	0,571986
1024	0,002984	0,003073	0,027664	0,028553	0,569693	0,573153
512	0,003050	0,003211	0,028557	0,029485	0,572382	0,576368
256	0,003301	0,003494	0,030058	0,030941	0,577638	0,582565

We tested our algorithm for three different data sizes. Why we chose these sizes and what is the difference between them, will be described in the performance evaluation part. In the performance evaluation we took similar sizes. Overall we decided to take a size of 4096 Bytes. That means that the number of parts is the number of the whole buffer size that shall be sent divided through the part size. One will think about why 8 KB is not the best when it looks like the best size on the paper, because it fully uses the buffer. Apparently this depends on the latency and bandwidth of the network. Sending smaller pieces more often can be better. Another point is that the pipelining in the algorithm gets better the smaller the parts are, because the earlier all processes can begin to work. For all performance evaluations later on we used a part size of 4 KB. This finishes up the cube algorithm and we will continue with a new approach.

6 Considering the Underlying Hardware (Mesh Algorithm)

6.1 Introduction

In the following we want to introduce our second algorithm [12]. In [12] they call it DOPL that stands for “Dimensional – Order Pipelined Broadcast”. We will stick with that and will call it DOPL in our work too. The approach differs a lot from the first. The main difference is that this algorithm is especially designed for a mesh-shaped underlying hardware. That means that the cores have to be arranged as a matrix. This can be seen in Figure 4.
As one can see the processors are clearly arranged in a matrix shape. The algorithm will take care that only processors that are neighbors will communicate with each other. In general the algorithm is round based and needs the same number of rounds as the one before. The order of the algorithm is the same, with a slightly different complexity. Therefore we do not want to fully explain this. For completeness we want to say that the exact lower bound for the algorithm can be found in [12]. The algorithm is adaptable to any mesh or torus structured hardware with some restrictions. The cores have to be bidirectional connected and the message passing

library has to support asynchronous send and receive functions. Both requirements are fulfilled on the SCC by the RCCE and iRCCE library.

6.2 The Algorithm itself

The algorithm sees the SCC as a matrix. The goal of the algorithm is to pipeline as much data as possible. The better the pipelining is the better is the performance in the end. The difficulty is how to pipeline the data among the cores that only cores who are neighbors have to communicate with each other to keep the latencies low. In [12] they present the following schema to handle this problem. We want to explain it via an example.

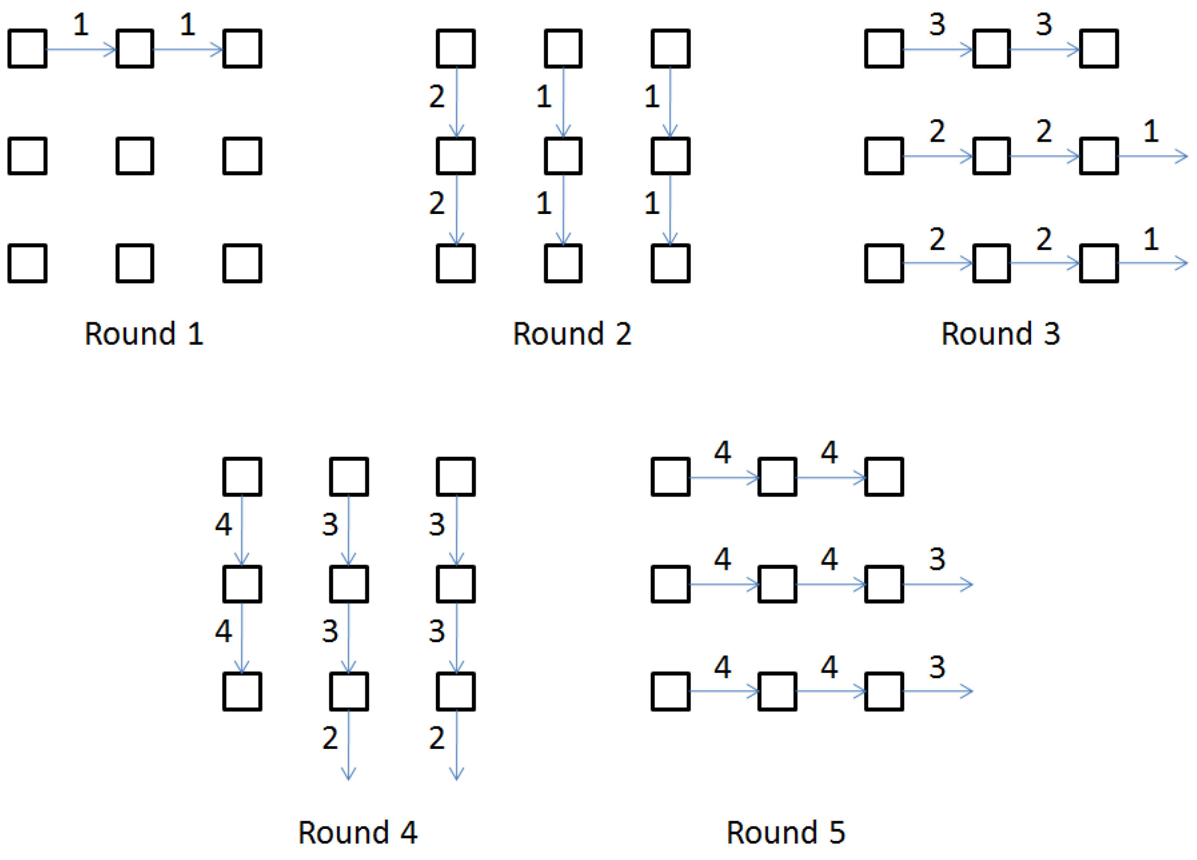


Figure 8, Schema of the DOPL Algorithm

The idea behind the algorithm is to pipeline a single part in a single round always in one direction. In the next round, the direction changes to cover the whole matrix of processes. In the example above the squares show processes and the arrows data movement. The process in the left upper corner is the source process (process 0) that has all the data. In the first round the data gets pipelined over the columns. Therefore process 0 sends the first part to his neighbor. His neighbor receives it and sends it to his neighbor and so on until all processes in the same row have received the first part. The other processes cannot do anything because they do not have any parts yet. When round two starts, the direction for the pipelining changes. Processes now pipeline over the rows. A process that starts the pipelining is called head. Therefore all processes in the first row are head processes. The same goes for tails. A tail is the last process in the pipeline. In this case, all processes in the last row. The source process brings in a new part to enable a constant pipelining without waiting for new parts. The other processes pipeline the first part, because they do not have any other. The direction changes again in round three, this time from rows back to columns. The source process pipelines the third part in his row and all other the second. For the first time a wraparound has to happen, because the processes in the first column do not have the first part received yet, except the source. Therefore all tail processes wrap around the first part to the head processes of this round. The wraparound is a costly factor, because it is not directly supported on the SCC. For the last two rounds the schema is the same as for the previous three rounds. The pipeline direction changes with every round and the processes keep sending data over the direction.

We want now to explain more detailed what a process or core has to do each round. We can say that the cores get mapped to different roles. A role is a description of what the core has to do this round. Every round each core calculates which role he plays this round. In some special cases a process can have two roles at the same time, if all requirements for the roles are met. All in all there are 4 different roles that we will describe now.

The Source:

The source is the first role we want to describe. This role is dedicated to exactly one core. It is impossible that more cores calculate out this role. Being the source is a special role because it never changes during the algorithm. In our case it is core 0, because core 0 is the core that has all the data.

The Head:

The source before was dedicated to a single process. In contrast to this the head role can be given to many processes. The heads are always the processes that start the pipelining. The pipelining goes column or row wise, depending on the actual round as described before. If for example process 13 is a head this round, then the pipelining goes through the whole row, which would be 13, 15, 17, 19, 21 and at last 23 according to Figure 4. In the end all these cores have the data from core 13. The head of course pipelines a part that he received in a previous round and that is new for the other processes in his pipeline. The head is always the first process in his dimension (row or column). A process in the middle cannot be the pipeline starter, because the pipelining is always over a full dimension to gain the maximum performance. The source has a special privilege because it is a head in each round.

The Tail:

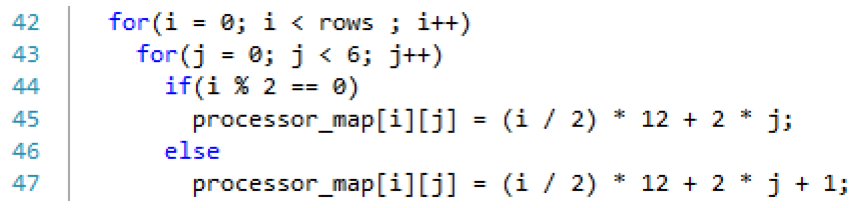
Each pipeline has a head and a tail. The tail is the last process in the pipeline. It only receives parts from his previous process to complete the pipeline. When the tail has the full data the pipeline is finished. After this the tail may send a part to the head if the head needs this. We will call it a wraparound. This can be necessary sometimes. We will describe later when these wraparounds can happen.

The Pipeliner:

The pipeliner is our last possible role. Every process in the middle of the matrix is a pipeliner, because it cannot be the head or the tail. Being the pipeliner is the simplest role, because the process receives a part from one of his neighbors and sends it to the next neighbor. That's it.

Further the algorithm maps a Cartesian tuple to each core. This is necessary, because the algorithm only deals with the position within the mesh not with the actual core ID. The tuple consists of the row and the column the core is in. For example, core 17 is in the 3rd row and in the 2nd column, as shown in Figure 4 before, when we count from 0 and start in the left bottom corner. We want now continue with our source code for further explanations.

The first snippet shows the creation of the process mapping by their coordinates. It is the first thing the algorithm needs to do.



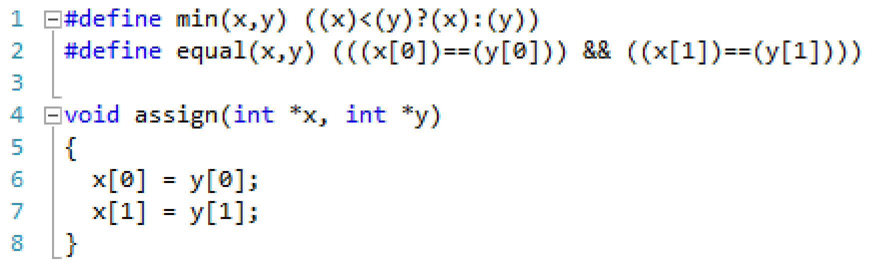
```
42    for(i = 0; i < rows ; i++)
43      for(j = 0; j < 6; j++)
44        if(i % 2 == 0)
45          processor_map[i][j] = (i / 2) * 12 + 2 * j;
46        else
47          processor_map[i][j] = (i / 2) * 12 + 2 * j + 1;
```

The variable $processor\_map$ above represents the mapping. The code above is especially designed for the structure of the SCC. That's why the constant values occur in the lines 45 and 47. The 12 in the code above can be seen as twice the columns. i gets set to the number of processor rows that one wants to use, to provide the algorithm for 12, 24, 36 or 48 cores. The j variable is always a full row.

If this array gets printed it would exactly look like this:

0	2	4	6	8	10
1	3	5	7	9	11
12	14	16	18	20	22
13	15	17	19	21	23
24	26	28	30	32	34
25	27	29	31	33	35
36	38	40	42	44	46
37	39	41	43	45	47

It may confuse that the mapping looks upside down compared to Figure 4 before. We did this, because it does not change anything in the algorithm and is easier to understand. Each core now has its exact position in the two dimensional mesh. This code above runs outside the main loop at the beginning of the program. The array needs to get built only once and does not change during the algorithm. Before we start with the main loop we have to mention some helper functions.



```c
#define min(x,y) ((x)<(y)?(x):(y))
#define equal(x,y) (((x[0])==(y[0])) && ((x[1])==(y[1])))

void assign(int *x, int *y)
{
  x[0] = y[0];
  x[1] = y[1];
}
```

The *min* define above returns the smaller one of the two parameters. The *equal* define returns true when both Cartesian tuples are equal and false if not. The *assign* function assigns a Cartesian tuple to another. Now we want to describe our main loop.

```
for(i = 0; i < kalt + d - 1; i++)
{
  imodd = i % d;
  assign(head, me);
  head[imodd] = src[imodd];
  assign(tail, head);

  tail[imodd] = (head[imodd] + m[imodd] - 1) % m[imodd];

  if(equal(me, head))
    if(equal(me, src))
      r = -1;
    else
    {
      r = i - d;
      kr = 1;
    }
  else
  {
    r = min(kalt - 1, i + level(head, src, imodd) - d);
    kr = k[imodd];
  }

  if(equal(me, tail))
    if(equal(head, src))
      s = -1;
    else
    {
      s = i - d;
      ks = 1;
    }
  else
  {
    s = min(kalt - 1, i + level(head, src, imodd) - d);
    ks = k[imodd];
  }

  r_buffer_size = (r == kalt - 1) ? last_size : kalt_size;
  s_buffer_size = (s == kalt - 1) ? last_size : kalt_size;
```

Before we want to explain the actual algorithm there is one important thing to mention. The data that shall be sent gets split up twice. The main division defines the number of total rounds together with the number of processors. It is the division that we showed in the example before and which is done in the code snippet above. The second division occurs when a single part needs to get passed throughout a row or a column, because the passing is implemented via a pipelining mechanism. The part gets split up by a pipeline size to perform the actual pipelining. We will describe that later.

The variable i represents the actual round. It starts by zero and goes to $kalt + d - 2$. $kalt$ is the number of the main parts that need to be sent and d represents the dimension of the matrix. In our case two. me is the Cartesian coordinates tuple of the own process. The other variables like $head$, $tail$ and src represent the different roles. They are Cartesian coordinate tuples too. Each process calculates the head and the tail of his row or column. The $imodd$ variable defines if the pipelining this turn is row wise or column wise. If it is 0 then the pipelining goes over the rows and if it is 1 then over the columns. There is one variable left for calculating the roles and this is m. m has the number of processes in the rows and columns as Cartesian tuple. It represents the length of the row and the column. This is needed for calculating the tail process. Every process now knows the head and the tail process of the row or column the pipelining goes over. In line 79 the processes begin to differ and to start with their roles. They check if their Cartesian coordinates match the Cartesian coordinates from the head, the tail or the source. The roles differ in which parts shall be sent or received. Each process calculates now which part shall be received during this round. The variable r shows that. r goes hand in hand with kr that determines in how many parts the sending buffer shall be split for the pipelining. kr is only necessary to differ from a wraparound if the process is a tail in this round. If it sends the wraparound then the part shall not be pipelined, because only one process receives it and that is the head. Therefore kr will be one, which means the whole buffer shall be sent. In all other cases kr will be $k[imodd]$, which is a constant that we will explain later. The exact same goes for the part that shall be sent. It is represented by s. As r before to kr, s is related to ks.

In some special cases r or s is -1, then the operation will be skipped. This is the case, when a tail tries to wraparound and the head is the source. The source of course does not need to receive any parts, because it has all data and the tail in this round does not need to send the wraparound to the source.

Each process checks now if he is the head, the tail, the source or just a normal process that passes data. The result shows in different values for r, kr, s and ks. When kr and ks normally pipe they will get the value of $k[imodd]$. We will talk about that later. There is one thing left and this is the $level$ function. The $level$ function is needed for all processes that pipeline. It determines which part needs to be sent. The calculation for the part that needs to be sent or received is $i + level - d$. The function gets called with the head, the source and the number of dimensions as parameters.

```
inline int level(int* a, int* b, int imodd)
{
  int i;

  for(i = 0; i < 2 && a[(imodd + i) % 2] == b[(imodd + i) % 2]; i++);

  return i;
}
```

The $level$ function returns in how many dimension the $head$ variable matches the src. In simpler words it says if the head in this round is the source then r and s will be one part higher, because the source brings in a new part every round as we described before in the example. If the head is not the source then r and s are one part lower in comparison to the pipeline where the source is the head.

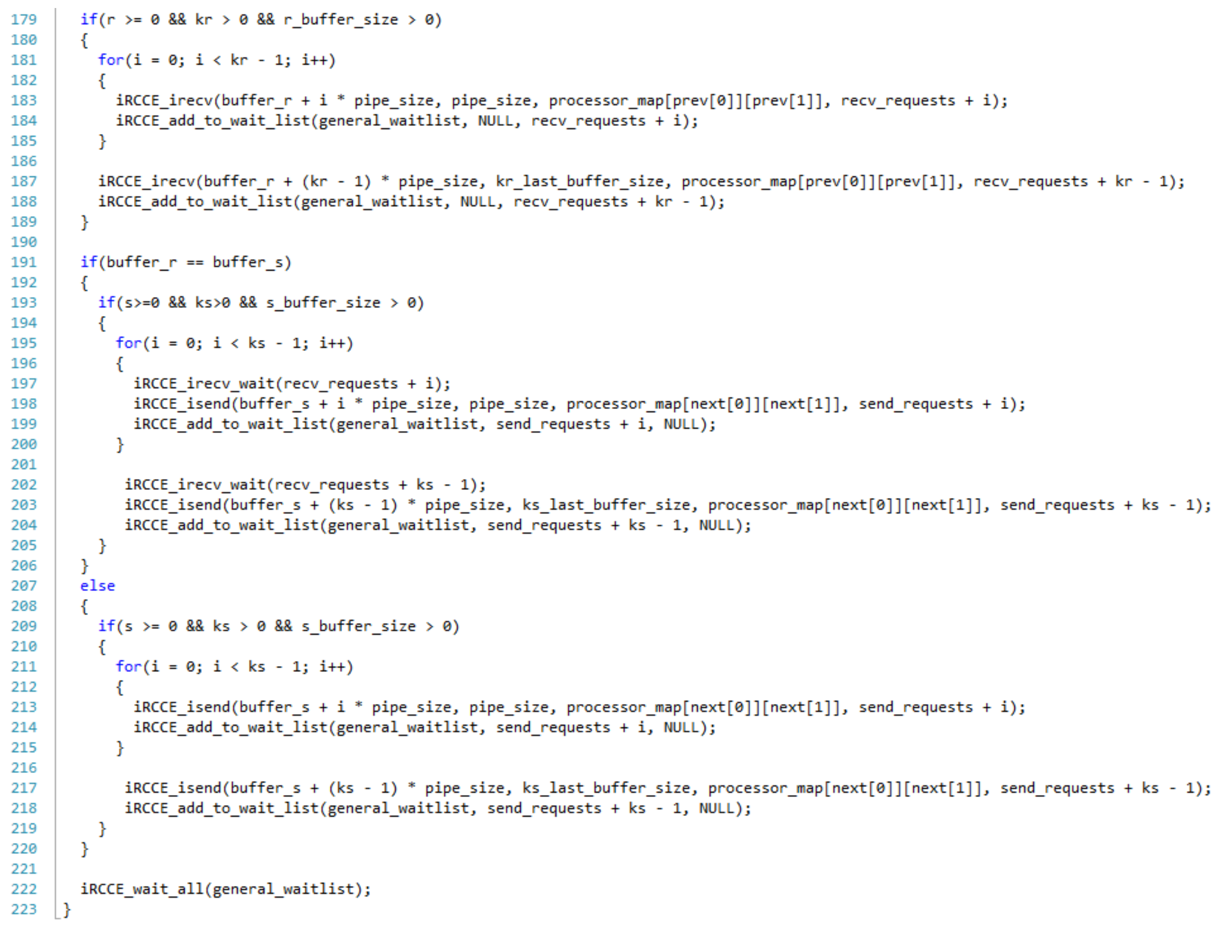
When all necessary parameters for the send and receive operations are calculated the processes call the pipeline function. The buffer size calculations in line 107 and 108 are for the last part, because this one may be smaller as the previous ones. r_buffer_size is the buffer size that shall be received. If r matches the last part, then the buffer size shall be the last size and not the general one. The same goes for the sending. We will continue with the pipeline function calls.

```
    if(r >= 0 && s >= 0)
      pipe(buffer + kalt_size * r, r, kr, buffer + kalt_size * s, s, ks, kalt, imodd, me,
        r_buffer_size, s_buffer_size, m, processor_map, &general_waitlist, recv_requests, send_requests);

    else if(r >= 0)
      pipe(buffer + kalt_size * r, r, kr, NULL, 0, 0, kalt, imodd, me, r_buffer_size, 0,
        m, processor_map, &general_waitlist, recv_requests, send_requests);

    else if(s >= 0)
      pipe(NULL, 0, 0, buffer + kalt_size * s, s, ks, kalt, imodd, me, 0,
        s_buffer_size, m, processor_map, &general_waitlist, recv_requests, send_requests);
  }
```

The pipeline calls differ if the processes want to send, receive or both. All values that are necessary for sending and receiving the right parts will be passed to the function. The last three parameters are only for the send and receive operations and have nothing to do with the algorithm, therefore we will skip that at the moment. The next code snippet shows the first part of the pipelining function.

```
131    int prev[2], next[2];
132    int i, kr_last_buffer_size = 0, ks_last_buffer_size = 0;
133
134
135    const int pipe_size = 2048;
136
137    assign(prev, me);
138    prev[imodd]=(prev[imodd] + m[imodd] - 1) % m[imodd];
139
140    assign(next, me);
141    next[imodd] = (next[imodd] + 1) % m[imodd];
142
143    if(r>=0 && kr && r_buffer_size > 0)
144    {
145      if(kr == 4)
146      {
147        kr = r_buffer_size / pipe_size;
148
149        if(kr * pipe_size != r_buffer_size)
150        {
151          kr++;
152          kr_last_buffer_size = r_buffer_size - (kr - 1) * pipe_size;
153        }
154        else
155          kr_last_buffer_size = pipe_size;
156      }
157      else
158        kr_last_buffer_size = r_buffer_size;
159    }
160
161    if(s >= 0 && ks && s_buffer_size > 0)
162    {
163      if(ks == 4)
164      {
165        ks = s_buffer_size / pipe_size;
166
167        if(ks * pipe_size != s_buffer_size)
168        {
169          ks++;
170          ks_last_buffer_size = s_buffer_size - (ks - 1) * pipe_size;
171        }
172        else
173          ks_last_buffer_size = pipe_size;
174      }
175      else
176        ks_last_buffer_size = s_buffer_size;
177    }
```

At the beginning of the function each process has to do some new buffer size calculations. As mentioned before, the buffer gets split up by the pipelining size. Our pipelining size is 2048 Bytes as we can see in line 135. We will describe later why it is this size. Each process calculates then the previous and the next process in the pipeline. This is done in the lines 137 – 141. After this the sending and receiving sizes get adapted to the pipelining size. The queries in line 145 and 163 may look a little bit awkward. When the pipelining goes the normal way, then the buffer size is 8192 Bytes for a single part. This part gets split up in 4 parts, each 2048 Bytes. That's why the kr and ks variable usually is 4. If it is not 4, then the part buffer shall be sent in one piece. This is the case when we have a wraparound. The following code shows the different send and receive operations.



```
179    if(r >= 0 && kr > 0 && r_buffer_size > 0)
180    {
181      for(i = 0; i < kr - 1; i++)
182      {
183        iRCCE_irecv(buffer_r + i * pipe_size, pipe_size, processor_map[prev[0]][prev[1]], recv_requests + i);
184        iRCCE_add_to_wait_list(general_waitlist, NULL, recv_requests + i);
185      }
186
187      iRCCE_irecv(buffer_r + (kr - 1) * pipe_size, kr_last_buffer_size, processor_map[prev[0]][prev[1]], recv_requests + kr - 1);
188      iRCCE_add_to_wait_list(general_waitlist, NULL, recv_requests + kr - 1);
189    }
190
191    if(buffer_r == buffer_s)
192    {
193      if(s>=0 && ks>0 && s_buffer_size > 0)
194      {
195        for(i = 0; i < ks - 1; i++)
196        {
197          iRCCE_irecv_wait(recv_requests + i);
198          iRCCE_isend(buffer_s + i * pipe_size, pipe_size, processor_map[next[0]][next[1]], send_requests + i);
199          iRCCE_add_to_wait_list(general_waitlist, send_requests + i, NULL);
200        }
201
202         iRCCE_irecv_wait(recv_requests + ks - 1);
203         iRCCE_isend(buffer_s + (ks - 1) * pipe_size, ks_last_buffer_size, processor_map[next[0]][next[1]], send_requests + ks - 1);
204         iRCCE_add_to_wait_list(general_waitlist, send_requests + ks - 1, NULL);
205      }
206    }
207    else
208    {
209      if(s >= 0 && ks > 0 && s_buffer_size > 0)
210      {
211        for(i = 0; i < ks - 1; i++)
212        {
213          iRCCE_isend(buffer_s + i * pipe_size, pipe_size, processor_map[next[0]][next[1]], send_requests + i);
214          iRCCE_add_to_wait_list(general_waitlist, send_requests + i, NULL);
215        }
216
217         iRCCE_isend(buffer_s + (ks - 1) * pipe_size, ks_last_buffer_size, processor_map[next[0]][next[1]], send_requests + ks - 1);
218         iRCCE_add_to_wait_list(general_waitlist, send_requests + ks - 1, NULL);
219      }
220    }
221
222    iRCCE_wait_all(general_waitlist);
223  }
```

For the algorithm we need the iRCCE library. It is important that we can use non – blocking send and receive functions. If we would use blocking functions the performance gain through the pipelining would be very little because of the synchronization between the processes.

The sending and receiving is very simple. If a process receives data, then it calls the asynchronous receive functions for it. He calls all receives for all pipeline parts, because they are non - blocking. This is done in the lines 179 – 189. As described in the iRCCE section before, every send and receive operation gets attached to a waitlist. The $add\_to\_wait\_list$ call is immediately after each send or receive operation. A process always receives from the previous one. This is shown in 183 and 187 by the parameter $\text{processor\_map}[\text{prev}[0][\text{prev}[1]]$. After this a process has to check if the part that he needs to send is the part that he shall receive this round. If it is, he of course needs to wait until the part is in his buffer and then starts sending. This is the first case of 191. In line 197 he waits for the part until the receiving is finished and then sends it to the next process in the line. If the part that he wants to send is a different one, then the process can just send it, like the lines from 209 to 220 show. Before the function gets closed, each process has to wait until all sending and receiving functions are finished. When the pipelining function is finished, the next round starts and the calculations for the roles and the other variables have to be done again.

6.3 Getting out the best Performance

It is difficult to choose the buffer size perfectly, because we have to split the buffer twice. One time for the number of parts and a second time for the pipelining. For the number of parts we take the same consideration as before. It will not make sense to choose a larger part size than 8 KB, because it will not fit into the message passing buffer. We could choose it smaller but then the pipelining will suffer from it. Thus we came to the conclusion that a part size of 8 KB should be perfect.

For the pipelining size we made a lot of benchmarks as the following table shows.

	50.000 Bytes	190.000 Bytes	1.900.000 Bytes
Pipe Size	Time	Time	Time
8102	0,001598654999626	0,006509817000452	0,109650387999570
4096	0,001059310999874	0,005065019999996	0,075469655999823
2048	0,000858949999947	0,004445106000258	0,065567735000289
1024	0,000925754000106	0,004538215999976	0,072228716999565
512	0,001181749999579	0,005174123000448	0,083540251999623
256	0,001639528999942	0,006994272999804	0,100209688999583

In the test we considered only buffer sizes that divide 8 KB. Other data sizes will waste some free space in the last part. As we can see in the table above the best size is 2 KB. That means that the part that shall be sent gets split up in 4 pieces and these pieces get pipelined. For all benchmarks later we used a pipelining size of 2 KB. That sums up the description of the mesh algorithm. In the next chapter we will discuss the performance evaluation of both algorithms.

7 Performance Evaluations

The performance evaluation will show if it is reasonable to implement an algorithm especially for a given hardware or if the advantage is not worth the effort in doing this. We tested the algorithms with three different kinds of data sizes. These sizes represent the different kinds of buffers. The smallest size fits in the Message Passing Buffer, the next one in the L2 Cache and the largest one, needs to be stored in the DDR 3 off – chip memory. We will further describe them throughout the chapter. Additionally we benchmarked the algorithms with 4 different numbers of cores for each data size. For the benchmarks we used the two implementations of the above described algorithms and the RCCE standard algorithm for broadcasting. The RCCE algorithm is not optimized in any kind, but shall be part of the measurements for completeness. We will start with the benchmarks for small data sizes.

7.1 Small Data Sizes

We consider data sizes as small as long as they fit in the Message Passing Buffer. That means that they are not larger than 8 KB.

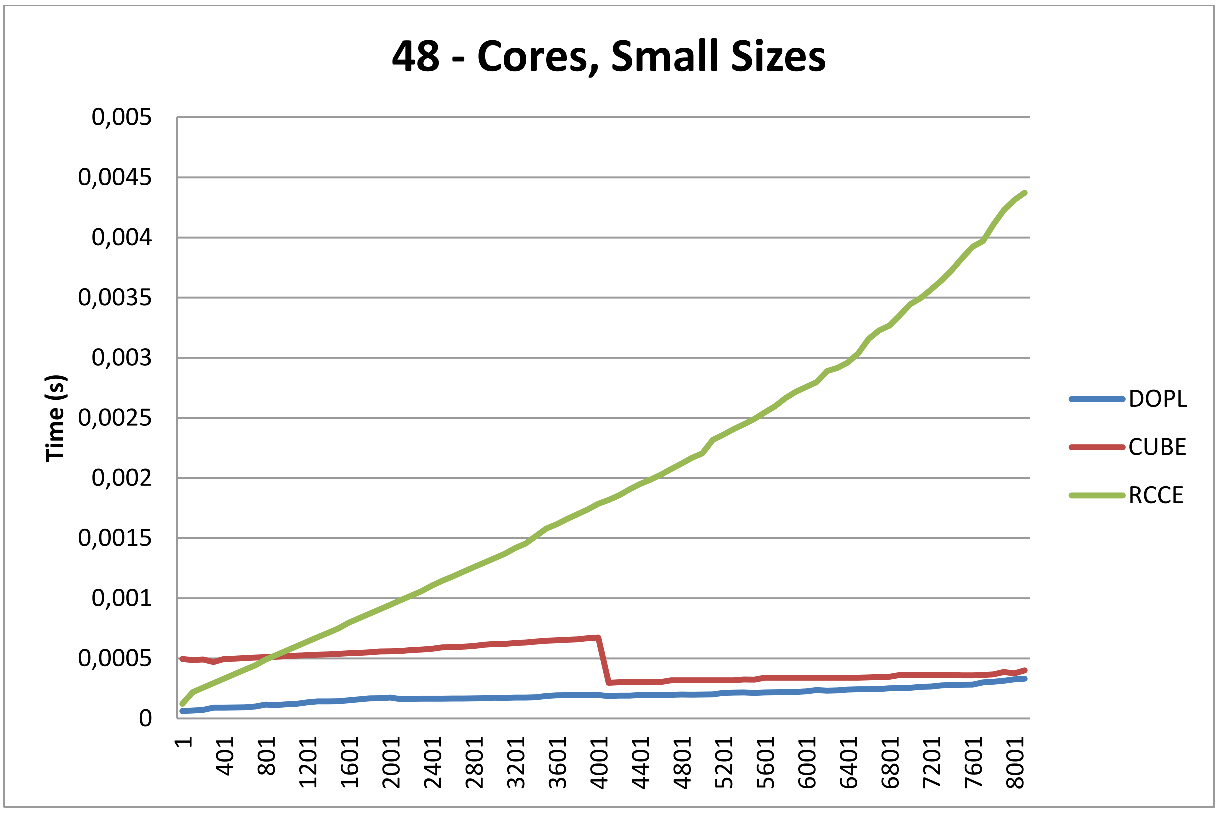


Figure 9, Broadcast of Small Data Sizes (1 – 8001) with RCCE

The starting buffer size is 1, and the buffer increases each round by a 100 until it reaches 8101 bytes. The measurement was repeated 1000 times for each size. We took the minimum runtime of the 1000 runs, because it is the runtime with the least interference and disturbance. One can see at first sight that the RCCE standard algorithm is very slow and does not scale well with the increasing buffer size. The gap between our algorithms and the RCCE algorithm is huge. We are nearly 10 times faster. In the figure above it is hard to see how much the difference between our two algorithms is. Therefore we remove the RCCE algorithm from the graphic.

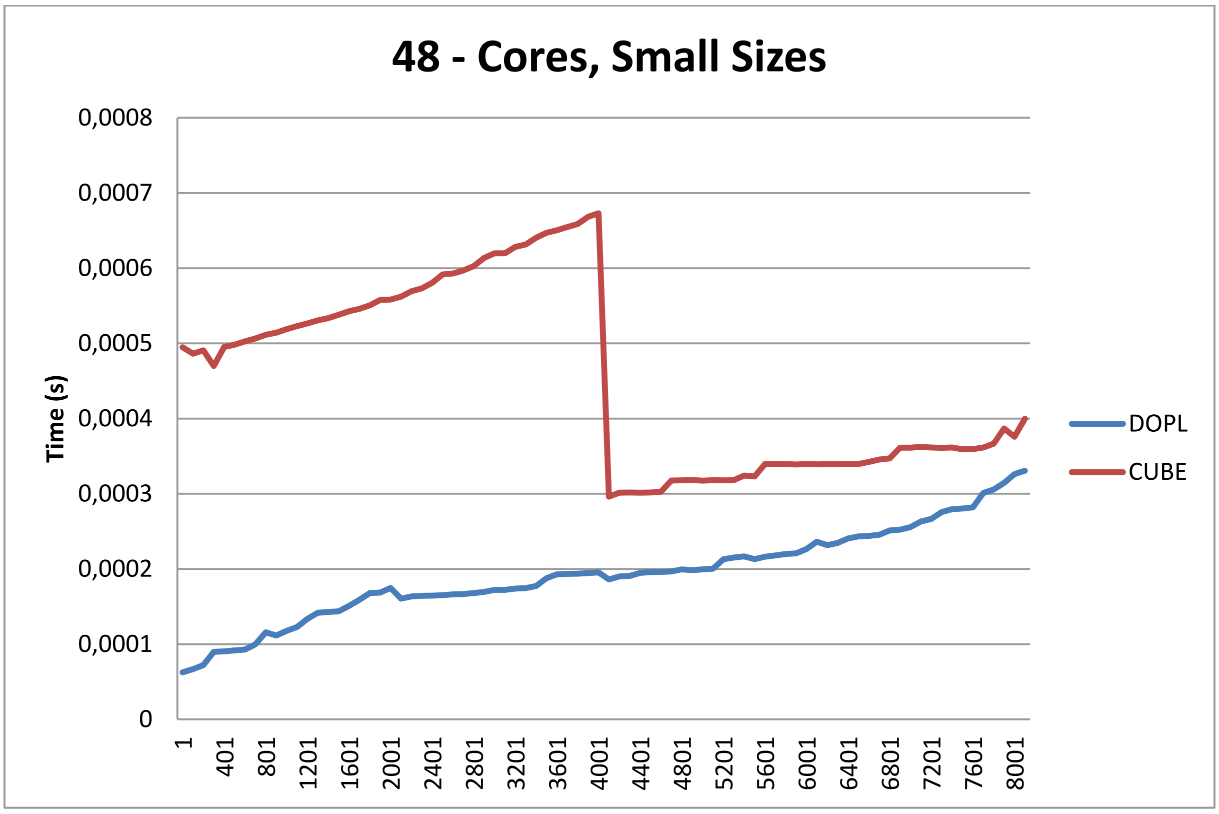


Figure 10, Broadcast of Small Data Sizes (1 – 8001), without RCCE

We can see now that the DOPL algorithm is about 20% faster than the Cube algorithm. The huge performance jump in the middle of the figure of the Cube algorithm is because it splits the sending buffer into two parts at a size of 4 KB or greater. This results in a better pipelining and work distribution over all cores. In the DOPL algorithm it is nearly the same, but the jump is not that large. At a size of 2 KB it also starts to split up the buffer. The Figure above only shows the runtime when all 48 cores are used, but it is interesting too how well the algorithms scale when we change the number of used cores. Therefore we performed another experiment. The result is shown in the next figure.

.

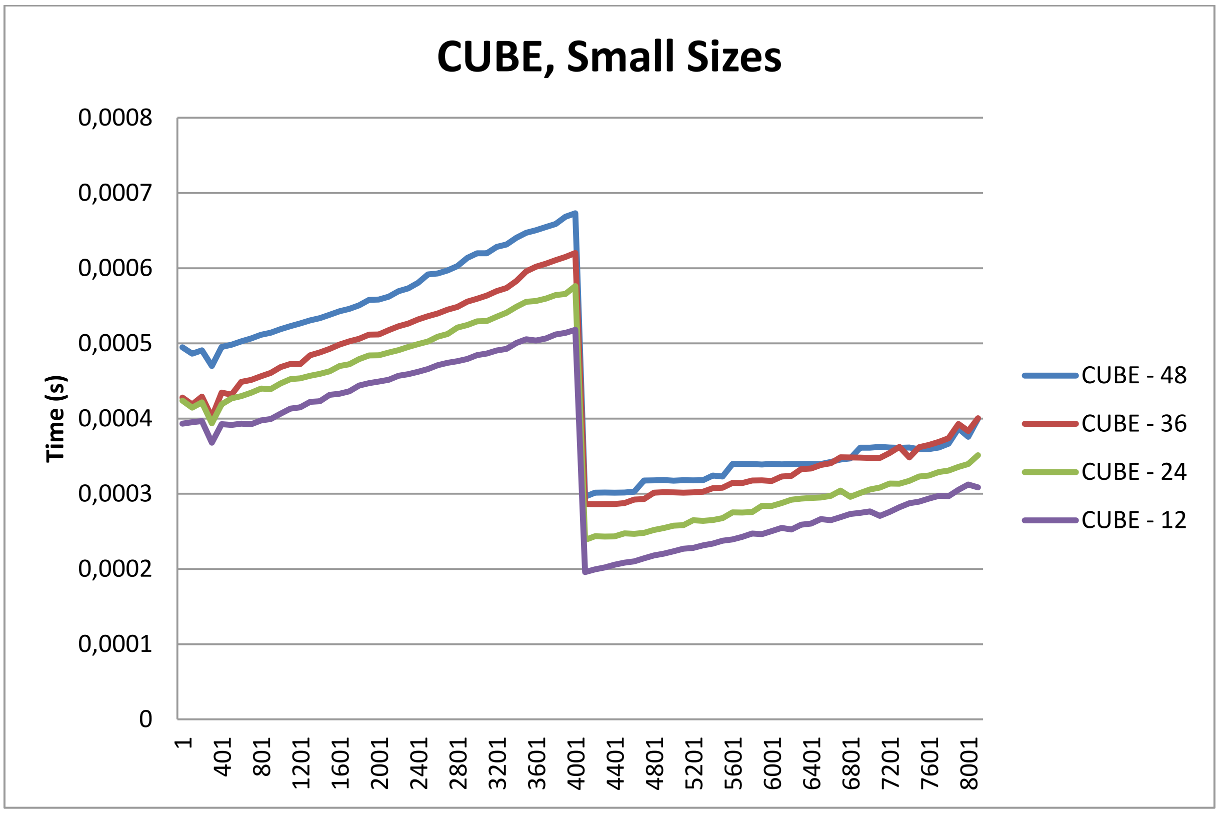


Figure 11, Broadcast of Small Data Sizes (1 – 8001), Cube Scaling

The figure above shows the Cube algorithm with 12, 24, 36 and 48 cores. As one can see it scales very well with the number of cores for small data sizes. It does not suffer much because the pipelining gets better with more cores and the algorithm tries to keep all cores busy. Therefore the critical path does not get much longer. The next figure shows the DOPL algorithm.

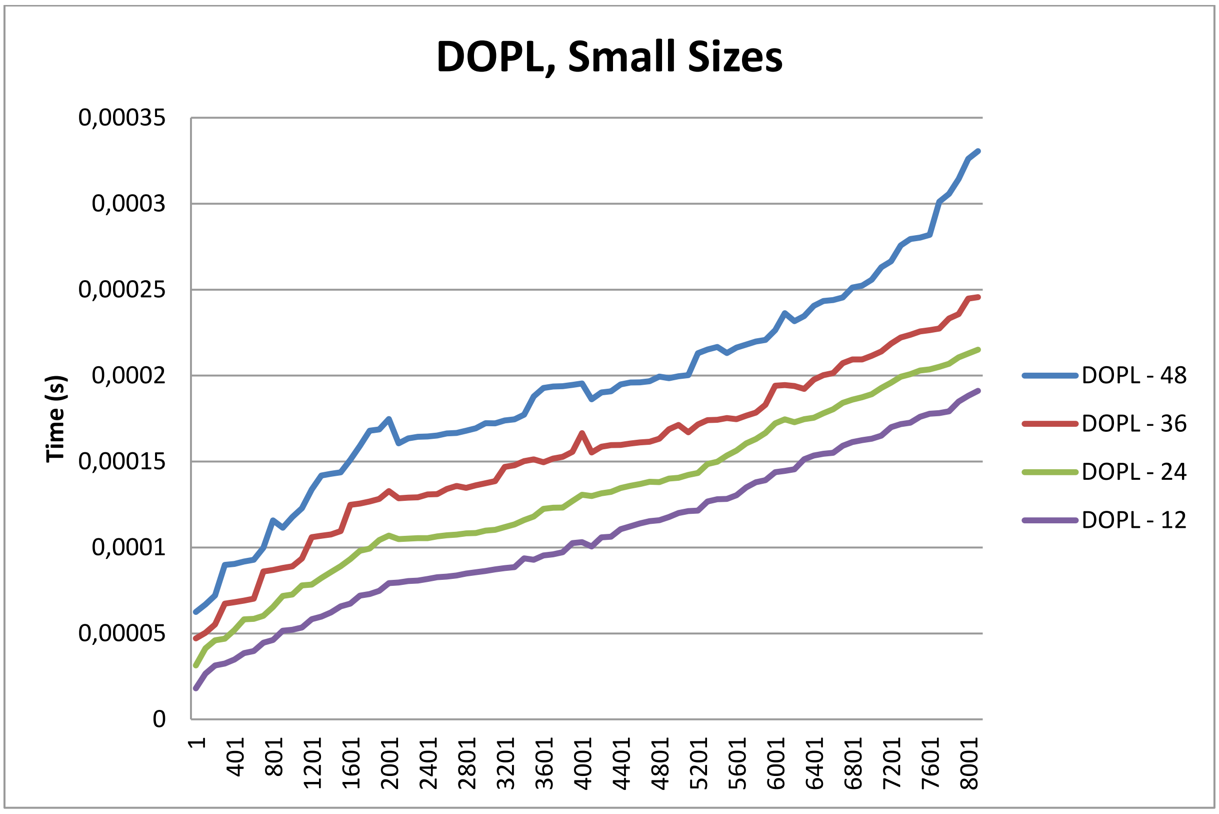


Figure 12, Broadcast of Small Data Sizes (1 – 8001), DOPL Scaling

The DOPL algorithm shows nearly the same picture as the Cube algorithm before. As we can see in the figure above, when we use 48 cores the runtime goes up faster than before, when the data size gets near to 8 KB. We are not sure why exactly this is the case. We suppose that the network gets overloaded when all cores are used and many parts are sent over the network at the same time, but we cannot proof it. Except this behavior the algorithm scales very well when we use more cores. The runtime of course goes up when we use more cores, but as we can see when we double the cores, the runtime does not nearly double.

7.2 Medium Data Sizes

Medium data sizes are sizes that do not fit in the Message Passing Buffer, but can be stored in the L2 cache of each core. The size of the buffer has to be larger than 8 KB and smaller than 256 KB. We started with a size of 10000 bytes and went to 190000 bytes with a step size of 5000 bytes.

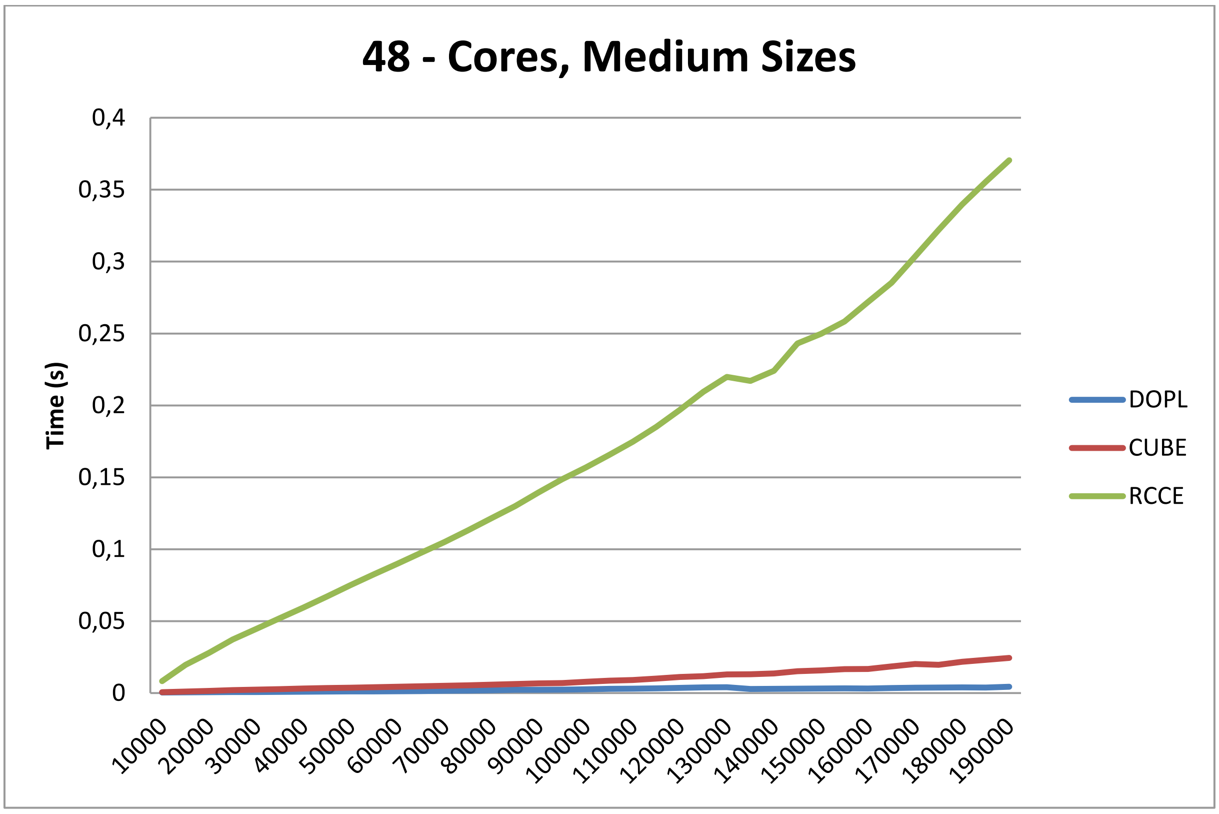


Figure 13, Broadcast of Medium Data Sizes (10k – 190k), with RCCE

The gap between our two algorithm implementations and the RCCE standard algorithm became very large. The RCCE algorithm needs about 10 times more time than the rest. To compare our two algorithms we exclude the RCCE algorithm again from the graphic.

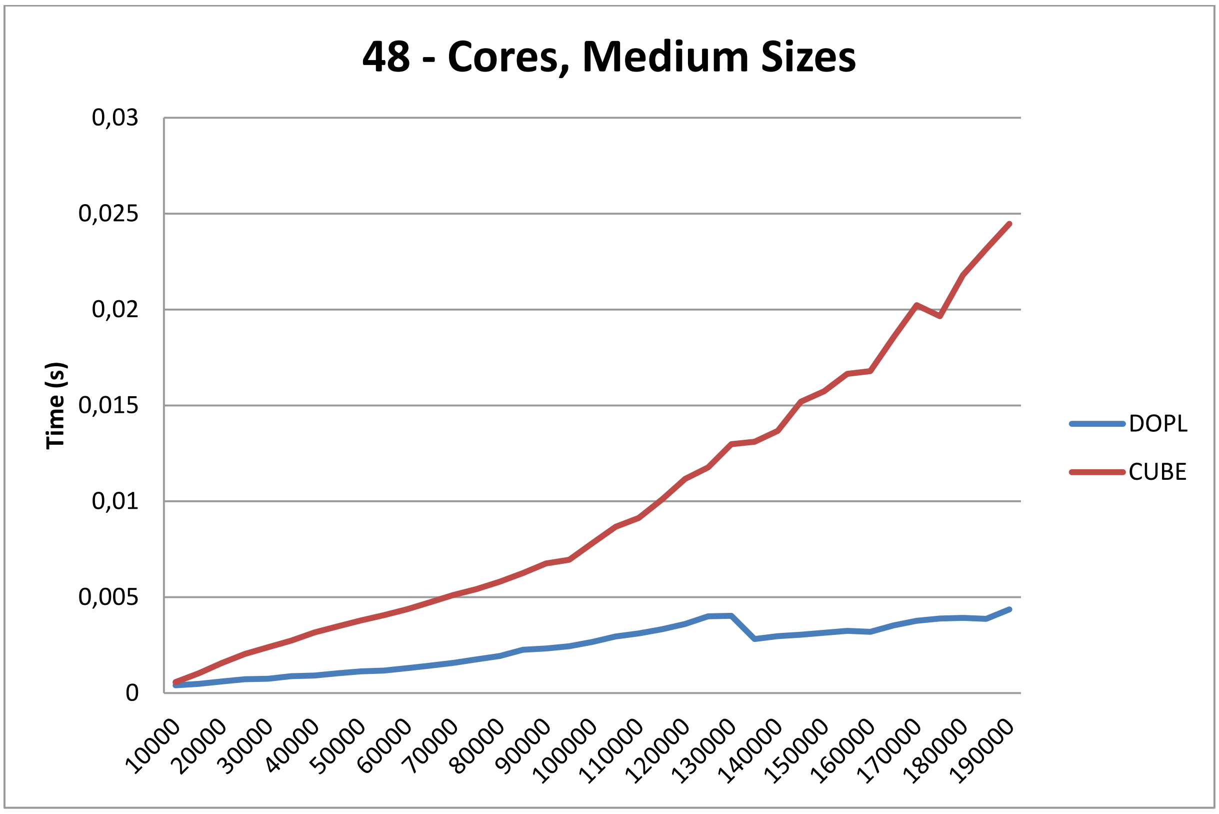


Figure 14, Broadcast of Medium Data Sizes (10k – 190k), without RCCE

As we can see in the figure above the gap between our two algorithms, compared to the small data sizes, increased too. This is mainly because the pipelining started to work well in the DOPL algorithm. Therefore the increasing data size can be handled easily. Our Cube implementation is about 5.5 times slower than the DOPL algorithm. It seems like that using the advantage of knowing the distances between the cores really pays off. The runtimes do not look as smooth as before, because there are a lot of buffer movements in it. Data can only be sent over the MPB of the cores. Therefore each core has to copy the part he wants to send from the L2 cache to his MPB. The same goes for receiving, but the other way round. In the following figures we will compare the scalability for both algorithms for medium data sizes.

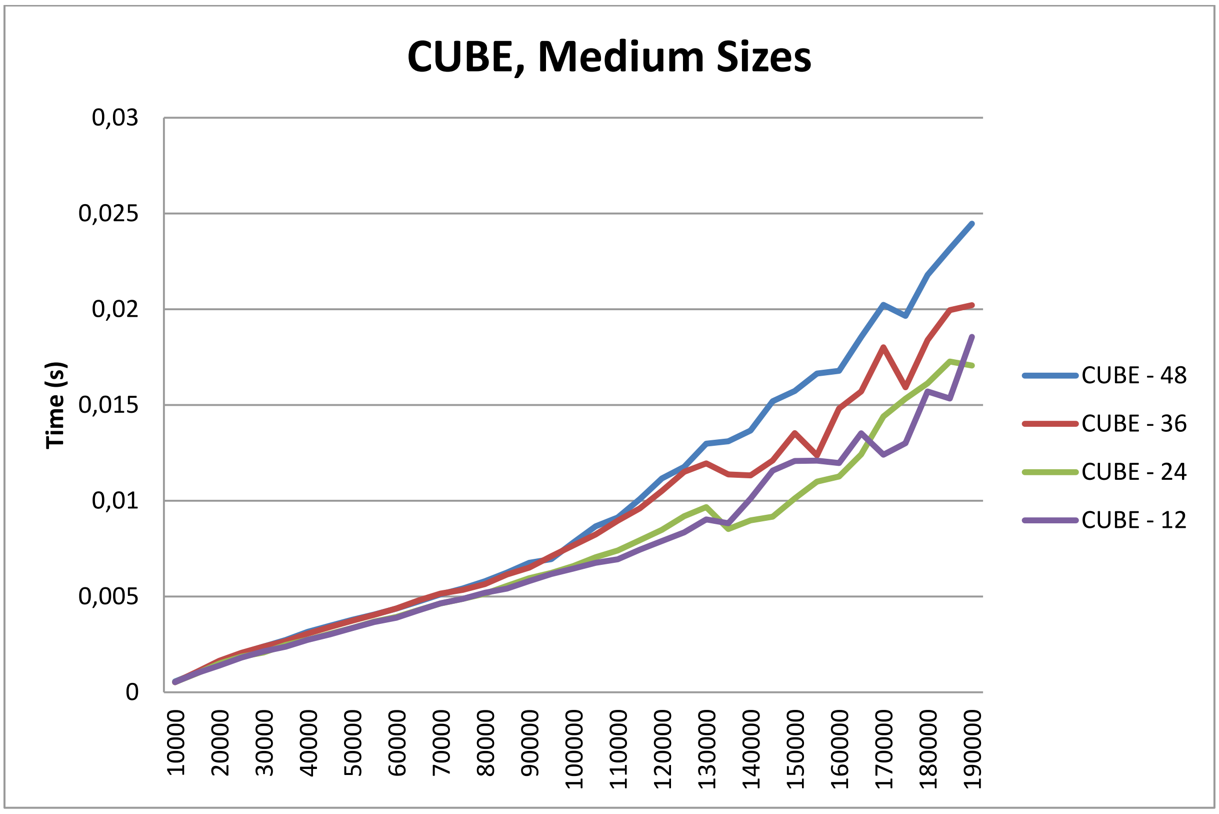


Figure 15, Medium Data Sizes, Cube Scaling

The scaling of the Cube algorithm looks nearly the same as for small data sizes before, with the same effect as before when we use all cores.

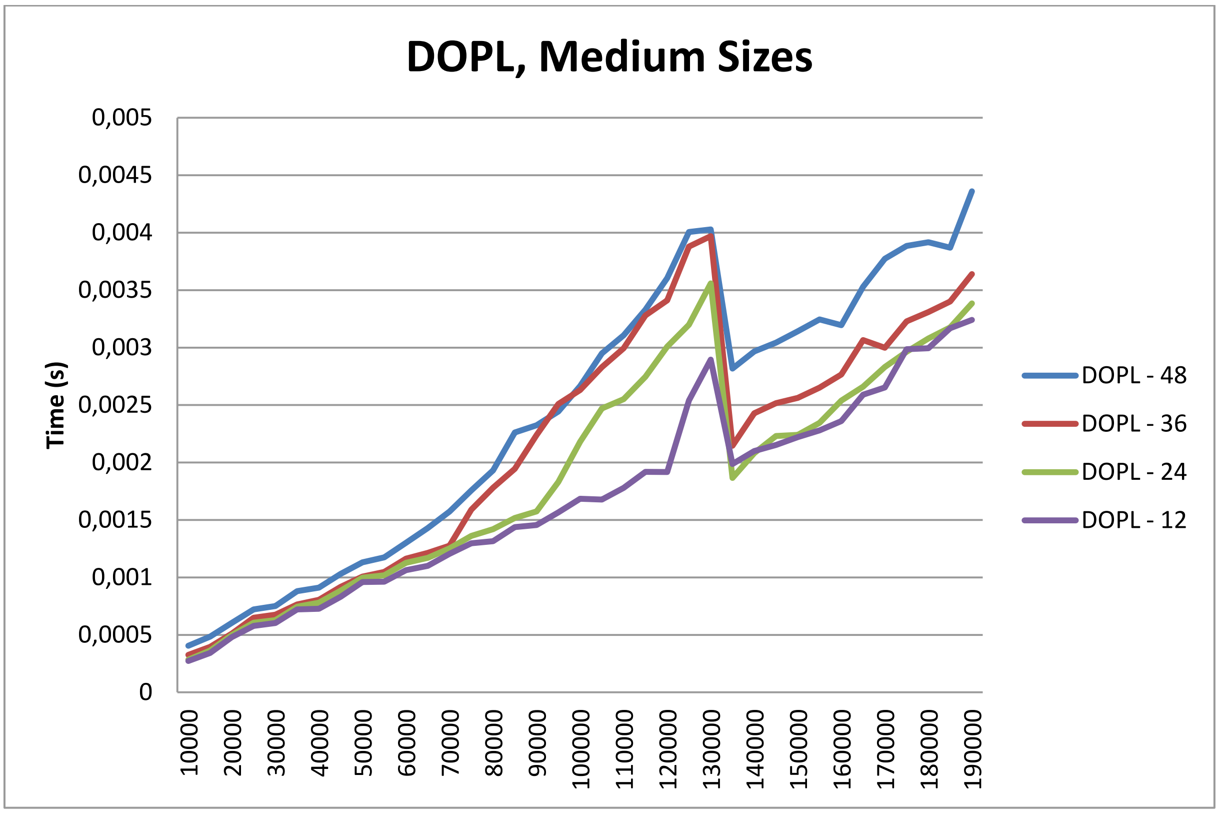


Figure 16, Broadcast of Medium Data Sizes (10k – 190k), DOPL Scaling

The DOPL algorithm shows the same picture again as before for small data sizes. The scaling is well again, but when all cores are used the performance slows down a bit.

7.3 Large Data Sizes

Large data sizes are sizes that do not fit in the L2 cache of the cores. They have to be transferred from the off – chip DDR 3 memory to the L2 caches. They may not be the best testing sizes, because the performance should suffer a lot from the access times of the DDR 3 memory. But it is still something that should be tested for a complete overview of the performance of the algorithms. The sizes start from 300000 bytes and will go up to 1.9 million bytes with a step size of 100000 bytes.

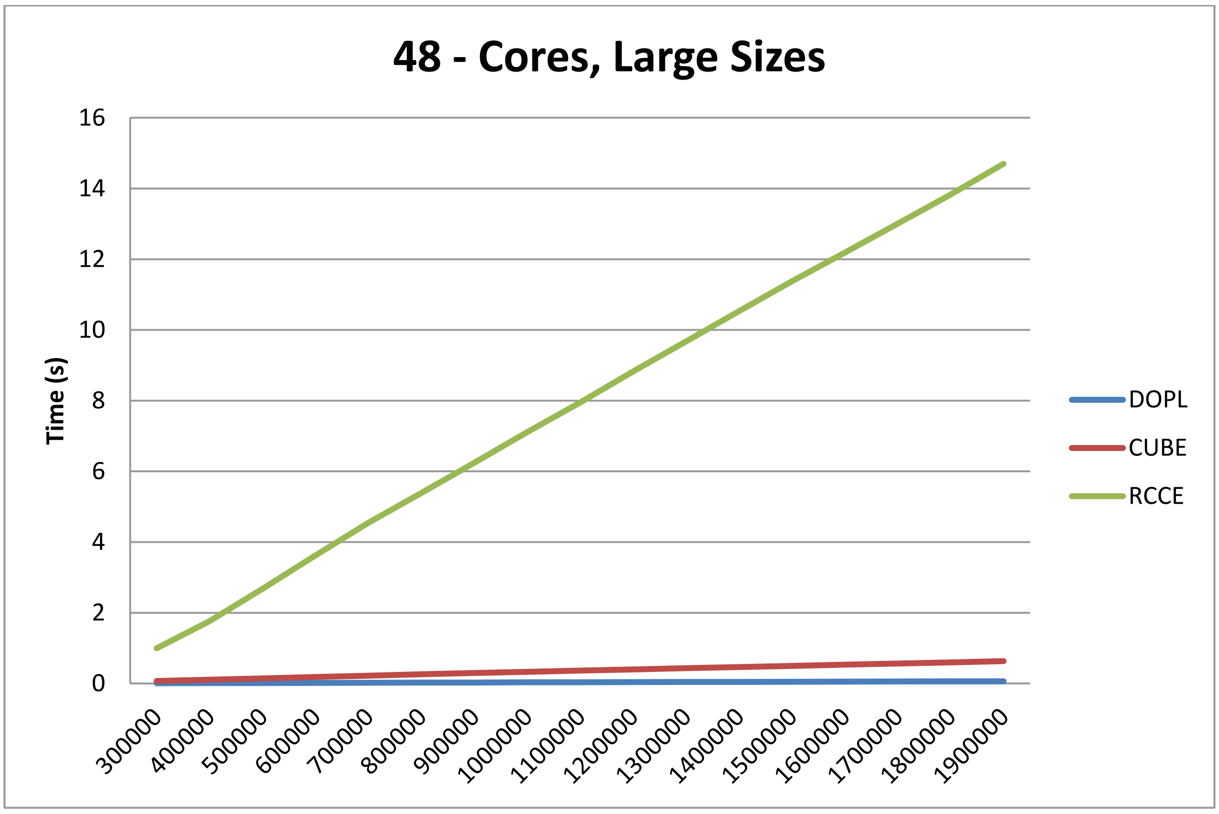


Figure 17, Broadcast of Large Data Sizes (300k – 1900k), with RCCE

The benchmark figure with the RCCE algorithm is just for completeness, because we cannot see any useful information that is related to the performance difference of our two algorithms, but it shows that both our algorithms really took off in comparison to the RCCE standard algorithm. In the following graphic we will remove the RCCE algorithm as before.

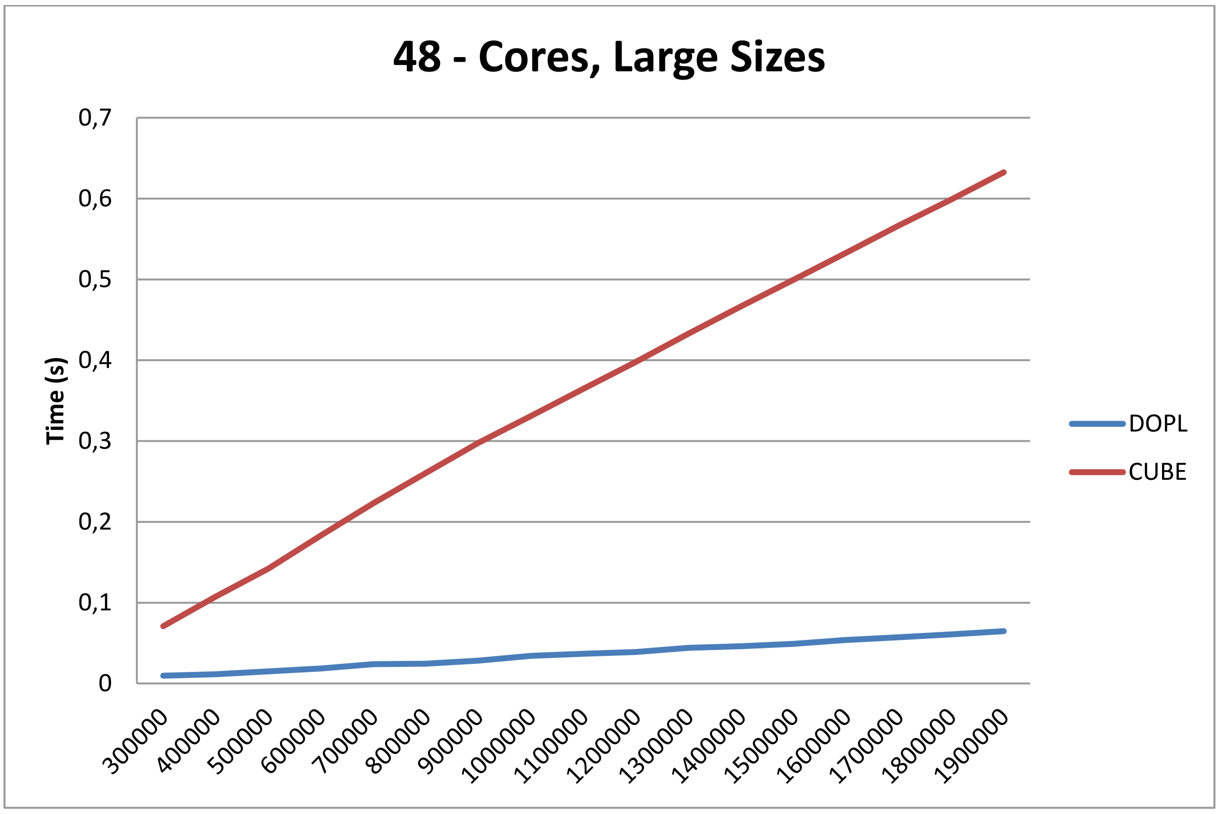


Figure 18, Broadcast of Large Data Sizes (300k – 1900k), without RCCE

The gap between our two algorithms grew larger. The DOPL algorithm only needs a tenth of the time the Cube algorithm needed. This is mainly because the larger the data size is the more parts are sent over the network and the bigger gets the advantage of the DOPL algorithm. The fear that the DDR 3 memory access times would kill the performance of the DOPL algorithm seems to be unfounded. The pipelining is good and the large data sizes are no problem to handle. In the following figures we will discuss the two algorithms with a different amount of cores.

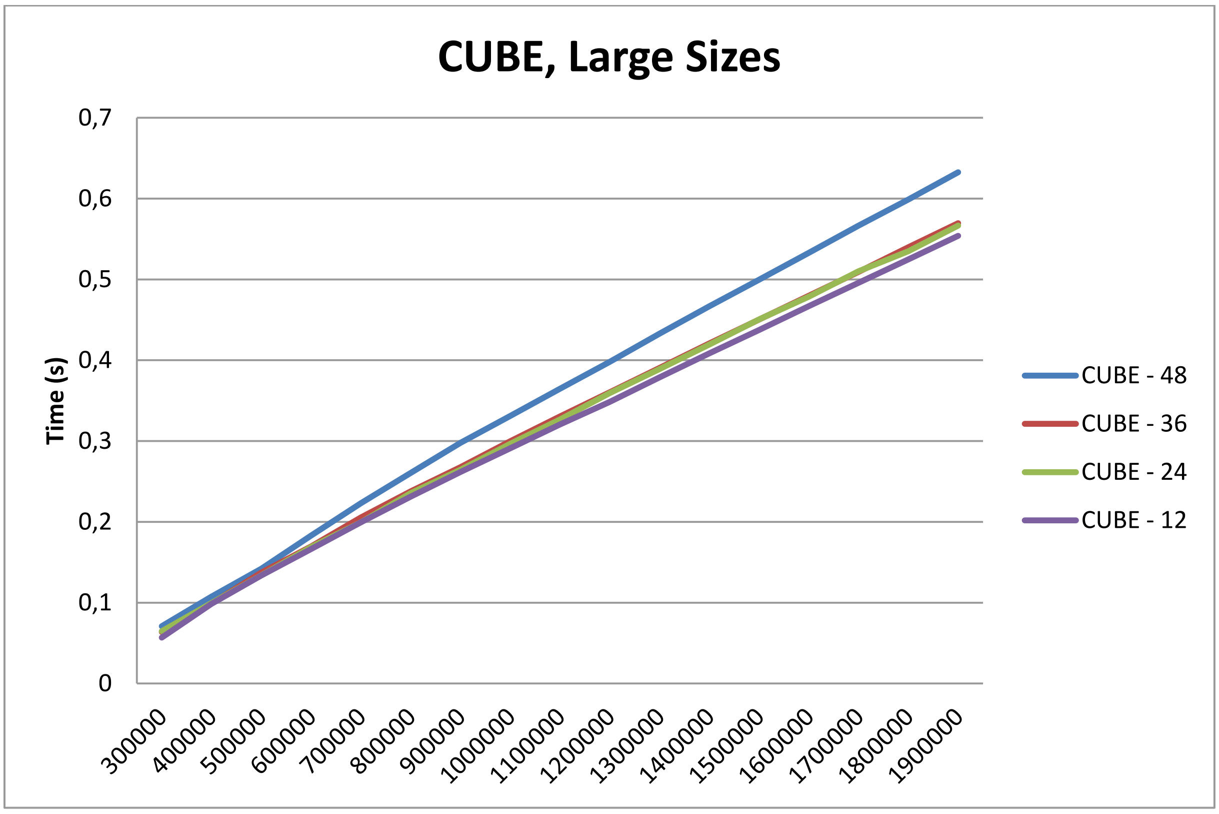


Figure 19, Broadcast of Large Data Sizes (300k – 1900k), Cube Scaling

As before, we will start with the performance of the Cube algorithm. The scaling looks very well, but when all cores are used the performance slows down, similar to the results of the DOPL algorithm before.

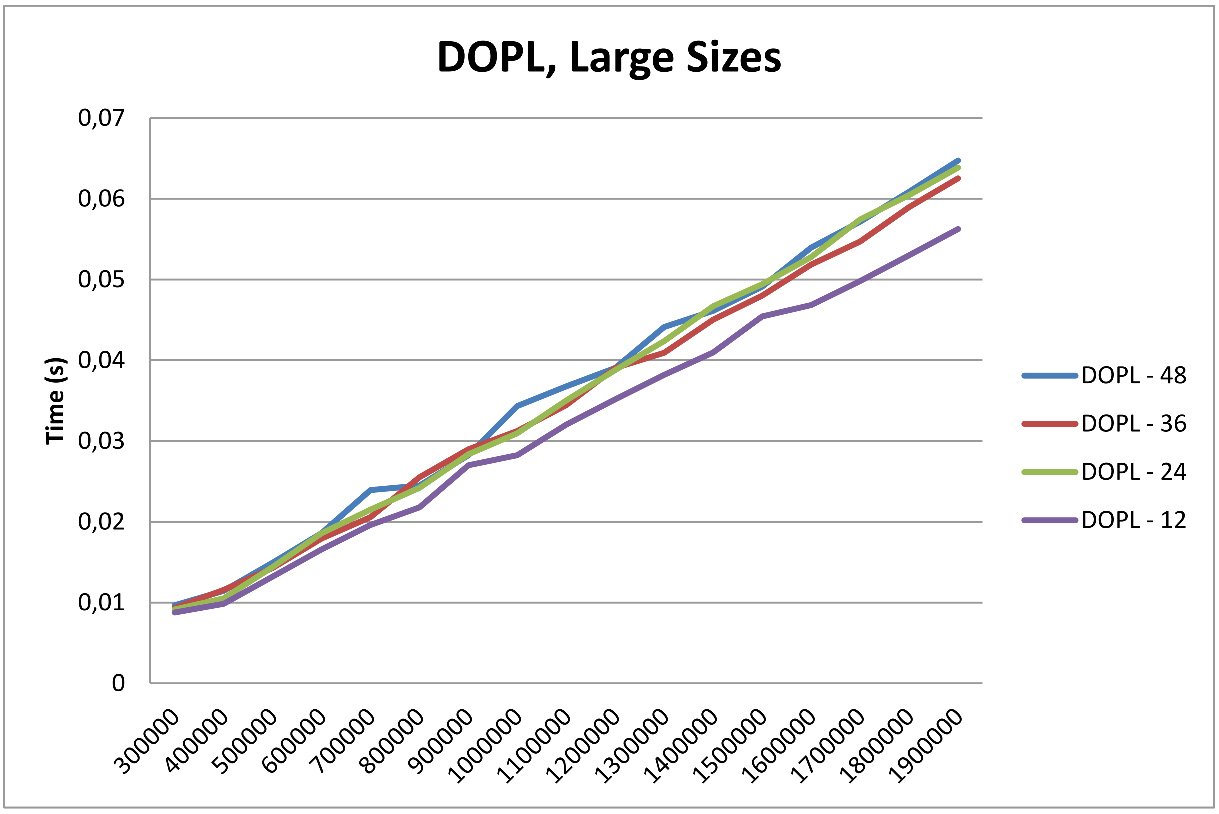


Figure 20, Broadcast of Large Data Sizes (300k – 1900k), DOPL Scaling

This time there is no outliner, when the cores increase. The network seems to handle the requirements quite well. It probably helps that the memory access times to the DDR 3 memory are quite long and the network does not get that flooded. The scaling for large data sizes is very good of the DOPL algorithm.

7.4 Performance Conclusion

Both algorithms showed a good performance over all benchmarks. They did not lack in performance when they used more and more cores. When we think about the data sizes, the DOPL algorithm gains more and more advantage the larger the sizes are. This is mostly because the buffers get split up in many single parts and each part takes the advantage of being transferred to a neighbor instead of somewhere else. All in all we can say that the benchmarks show the same results as the PingPong test. The DOPL algorithm outperforms the Cube algorithm by far, especially for medium and large data sizes.

8 Final Conclusions

To answer the question if the implementation of a broadcast algorithm for a given hardware is useful, we want to say that this cannot be said in general. The DOPL algorithm performed very well and outperformed the CUBE algorithm by far. When we look closer on the hardware, we have to say that the SCC is an accommodation for the CUBE algorithm. This is because all cores are on one chip. This has the advantage that the latency is minimal compared to for example a cluster. Therefore the punishment for communicating with processors that are far away does not hurt that much. We can say that the performance gap increases with the distance between the cores. Another point is that a broadcast is most of the times a part of another algorithm. Therefore it depends a lot on this algorithm how often he uses a broadcast. Finally we want to say that is very dependent on the circumstances if it is useful to implement a broadcast for a specific hardware, but it definitely has potential. Even on the SCC we showed that an especially adapted algorithm outperforms a general one by far.

9 References

[1] Tony Bradley 2009, “Intel 48-Core Single-Chip Cloud Computer Improves Power Efficiency”, http://www.pcworld.com/article/183653/Intel_48Core_SingleChip_Cloud_Computer_Improves_Power_Efficiency.html, last visited 2013.06.30

[2] Tim Mattson 2011, “Many core processors at Intel: lessons learned and future work”, http://communities.intel.com/servlet/JiveServlet/previewBody/19204-102-1-22473/SCC-manycore-lessons-and-futures.pdf, last visited 2013.06.30

[3] Mani Azimi, Naveen Cherukuri, D. N. Jayasimha, Akhilesh Kumar, Partha Kundu, Seungjoon Park, Ioannis Schoina and Aniruddha S. Vaidya, “Integration Challenges and Tradeoffs for Tera-scale Architectures “, Intel Technology Journal, Volume 11 Issue 03, 2007

[4] Intel Corporation 2010, “SCC External Architecture Specification (EAS)”, http://communities.intel.com/servlet/JiveServlet/previewBody/5852-102-1-9012/SCC_EAS.pdf, last visited 2013.06.30

[5] Tim Mattson and Rob van der Wijngaart 2011, “RCCE: a Small Library for Many-Core Communication”, http://communities.intel.com/servlet/JiveServlet/previewBody/5628-102-3-22522/RCCE_Specification.pdf, last visited 2013.06.30

[6] Message Passing Interface Forum, “MPI: A Message-Passing Interface Standard”, http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, last visited 2013.06.30

[7] Thomas Rauber and Gundula Rünger, “Parallel Programming for Multicore and Cluster Systems”, 2010

[8] Carsten Clauss, Stefan Lankes, Thomas Bemmerl,
Jacek Galowicz, and Simon Pickartz 2011, “iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library
for the Intel Single-Chip Cloud Computer”, http://www.lfbs.rwth-aachen.de/publications/files/iRCCE.pdf, last visited 2013.06.30

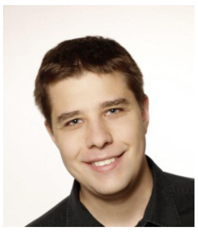
[9] Bin Jia, “Process cooperation in multiple message broadcast“, Parallel Computing, Volume 35 Issue 12, 2009

[10] S. Lennart Johnsson and Ching – Tien Ho, “Optimum Broadcasting and Personalized Communication in Hypercubes”, IEEE Transactios on Computers, VOL. 38, NO. 9, 1989

[11] Jesper Larsson Träff and Andreas Ripke, “Optimal Broadcast for Fully Connected Networks”, Journal of Parallel and Distributed Computing Volume 68 Issue 7 Pages 887-901, 2005

[12] Jerrell Watts and Robert A. van de Geijn, “A Pipelined Broadcast for Multidimensional Meshes”, Parallel Processing Letters p. 281-292, 1995

10 Curriculum Vitae

ANGABEN ZUR PERSON	Pichler Markus Alexander
	pichl0r@gmail.com Staatsangehörigkeit österreichisch
BERUF	Software Entwickler

BERUFSERFAHRUNG

01. Juli 2009 – 31. Mai 2012	**Fitness Trainer** Club Danube Adolf Schärf Platz 4, 1220 Wien (Österreich) Trainingsplanerstellung für Kunden Übungserklärungen Verkauf von Getränken an der Fitbar
01. Oktober 2009 – 30. Juni 2012	**Tutor für Einführung in die Programmierung und Algorithmen und Datenstrukturen** Universität Wien Dr. Karl-Lueger-Ring 1, 1010 Wien (Österreich) Verbesserung der Hausübungen der Studenten Fragenbeantwortung im Forum Tutorstunden an der Universität mit Betreuung der Studenten Tätigkeitsbereich oder Branche Informatik
02. November 2012 – Heute	**Software Entwickler** TRICENTIS Technology & Consulting GmbH Leonard-Bernstein-Straße 10 AT-1220 Wien Wien (Österreich) Umsetzung von Softwareprojekten in Microsoft C# Einbringung von Ideen zur Verbesserung der Software Zusammenarbeit mit anderen Entwicklern in einem Team

SCHUL- UND BERUFSBILDUNG

01. Oktober 2011 – Heute	**Diplom Ingenieur in Informatik (Scientific Computing)** Universität Wien,
01. Oktober 2006 – 06. September 2011	**Bachelor mit Auszeichnung in Informatik (Scientific Computing)** Universität Wien,

01 – 06 **Matura mit gutem Erfolg**
HTL Donaustadt, Abteilung EDV und Organisation,

97 – 01 **Allgemeine Grundausbildung**
Hauptschule II Wolkersdorf,

93 – 97 **Allgemeine Grundausbildung**
Volksschule Pillichsdorf,

PERSÖNLICHE FÄHIGKEITEN

Muttersprache(n) Deutsch

Weitere Sprache(n)

	VERSTEHEN		SPRECHEN		SCHREIBEN
	Hören	Lesen	An Gesprächen teilnehmen	Zusammenhängendes Sprechen	
Englisch	C1	C1	B2	B2	B2

A1/A2: elementare Sprachverwendung - B1/B2: selbstständige Sprachverwendung - C1/C2: kompetente Sprachverwendung
Gemeinsamer Europäischer Referenzrahmen für Sprachen

Kommunikative Fähigkeiten
- Teamgeist (Viele unterschiedliche Projektgruppen an der Universität)
- Guter Umgang mit Menschen (3 Jahre Tutor an der Universität und fast 3 Jahre Fitnesstrainer)

Organisatorische und Management Fähigkeiten
Organisationsfähigkeiten (Projektentwicklung Lehrveranstaltung an der Universität + zahlreiche Projekte an der Universität)

Computerkenntnisse
Betriebssysteme: Windows, Linux

Programmier Fähigkeiten: C++, C#, Java, Matlab, SQL, Fortran, XML, PHP, HTML

Multicore Programmierung: MPI, OpenMP

Grafikkarten Programmierung: OpenCL

Sonstige: Word, Excel, Powerpoint, UML

11 Appendix

10.1 PingPong Benchmark

```c
#include <string.h>
#include <stdio.h>
#include "RCCE.h"
#include <math.h>
#include "iRCCE.h"


#define MAX_SIZE 8192
#define START_SIZE 256
#define STEP_SIZE 256
#define ROUNDS 10000

#define WRITE_TO_FILE 1

int RCCE_APP(int argc, char **argv)
{
  int ME, YOU, i, j;
  int x,k,size;
  double timer = 0.0;
  int processors[5] = {1, 2, 10, 37, 47};
  int partner;
  int dummy=0;
  //double avg_time[5][(MAX_SIZE-START_SIZE)/STEP_SIZE+1];
  double total_time[5][(MAX_SIZE-START_SIZE)/STEP_SIZE+1];
  int sizes[(MAX_SIZE-START_SIZE)/STEP_SIZE+1];
  char *buffer;

  RCCE_init(&argc, &argv);
  iRCCE_init();

  ME = RCCE_ue();

  RCCE_barrier(&RCCE_COMM_WORLD);

  for(i = 0, size = START_SIZE; size <= MAX_SIZE; i++, size += STEP_SIZE)
  {

    sizes[i] = size;

    buffer = (char*)malloc(size * sizeof(char));

    for(k = 0; k < size; k++)
      dummy += buffer[k];

    if(ME == 0)
    {
      for(k = 0; k < size; k++)
        buffer[k] = k % 127;
      printf("\nSIZE: %d\n", size);
    }

    for(j = 0; j < 5; j++)
    {
      total_time[j][i] = 0.0;

      partner = processors[j];

      RCCE_barrier(&RCCE_COMM_WORLD);

      for(x = 0; x < ROUNDS; x++)
      {
        if(ME == 0)
        {
          RCCE_send(buffer, size, partner);
          RCCE_recv(buffer, size, partner);
        }
        else
        {
          if(ME == partner)
          {
            RCCE_recv(buffer, size, 0);
            RCCE_send(buffer, size, 0);
          }
        }
        if(x == 0)
          timer = RCCE_wtime();
      }

      timer = RCCE_wtime() - timer;
      total_time[j][i] = timer;

      if(ME==0)
        printf("Process: %d\ttime: %.15f\n",processors[j],timer);

      if(ME==processors[j])
        for(k=0;k<size;k++)
          if(buffer[k]!= k%127)
            printf("ERROR");
    }
    free(buffer);
  }

  if(ME==0)
    if(WRITE_TO_FILE)
    {
      FILE *fp;

      char name[100];

      for(j=0;j<5;j++)
```

```
        for(j=0;j<5;j++)
        {
          sprintf(name,"/shared/pichler/BENCHMARKS/PINGPONG/%d_%d_%d_%d_%d.txt",
              processors[j], START_SIZE, STEP_SIZE, MAX_SIZE, ROUNDS);

          fp=fopen(name,"w");
          for(i=0; i<(MAX_SIZE-START_SIZE)/STEP_SIZE+1; i++)
            fprintf(fp, "%d\t%1.9lf\t%1.9lf\n", sizes[i], total_time[j][i],
              total_time[j][i]/(ROUNDS-1));

          fclose(fp);
        }
      }

    RCCE_barrier(&RCCE_COMM_WORLD);

    RCCE_finalize();

    return 0;
}
```

10.2 Main Program for all Benchmarks

```c
#include <string.h>
#include <stdio.h>
#include "RCCE.h"
#include <math.h>
#include "iRCCE.h"
#include "HYPERCUBE.h"
#include "DOPL.h"
#include <stdlib.h>

#define MAX_SIZE 190000
#define START_SIZE 10000
#define STEP_SIZE 5000
#define ROUNDS 200
#define PRINTS 50


#define WRITE_TO_FILE 1

int RCCE_APP(int argc, char **argv)
{
  int ME, i, j;
  int x,k,size, cores;
  double timer = 0.0;

  //DOPL == 0, CUBE == 1, RCCE == 2
  double avg_time[3][(MAX_SIZE-START_SIZE)/STEP_SIZE+1];
  double min_time[3][(MAX_SIZE-START_SIZE)/STEP_SIZE+1];
  int sizes[(MAX_SIZE-START_SIZE)/STEP_SIZE+1];
  char *buffer;

  RCCE_init(&argc, &argv);
  iRCCE_init();

  ME = RCCE_ue();
  cores = RCCE_num_ues();


  RCCE_barrier(&RCCE_COMM_WORLD);

  for(i=0,size=START_SIZE;size<=MAX_SIZE;i++,size+=STEP_SIZE)
  {

    sizes[i] = size;

    buffer = (char*)malloc(size*sizeof(char));

    if(ME==0)
      printf("\nSIZE: %d\n", size);

    min_time[0][i] = 9999;
    avg_time[0][i] = 0;

    min_time[1][i] = 9999;
    avg_time[1][i] = 0;

    min_time[2][i] = 9999;
    avg_time[2][i] = 0;

    for(x=0;x<ROUNDS;x++)
    {
      if(ME == 0)
        if(x%PRINTS == 0)
          printf("Round: %d\n", x);

      for(j=0;j<size;j++)
        if(ME==0)
          buffer[j] = j%127;
        else
          buffer[j] = 'Z';

      RCCE_barrier(&RCCE_COMM_WORLD);

      if(ME == 0);
        timer = RCCE_wtime();

      dopl(buffer, size);

      RCCE_barrier(&RCCE_COMM_WORLD);

      if(ME==0)
        if(x%PRINTS == 0)
          printf("DOPL finished\n");

      if(ME == 0)
      {
        timer = RCCE_wtime() - timer;
        avg_time[0][i] +=timer;
        if(timer < min_time[0][i])
          min_time[0][i] = timer;
      }

      for(j=0;j<size;j++)
        if(buffer[j] != j%127)
        {
          printf("%d DOPL-ERROR in Position %d\tBuffer: %d, Expected: %d\n",
              ME, j,buffer[j],j%127);
          break;
        }

        for(j=0;j<size;j++)
          if(ME==0)
            buffer[j] = j%127;
          else
            buffer[j] = 'Z';

        RCCE_barrier(&RCCE_COMM_WORLD);

        if(ME == 0);
          timer = RCCE_wtime();

        HYPERCUBE_bcast(buffer, size);

        RCCE_barrier(&RCCE_COMM_WORLD);

        if(ME==0)
          if(x%PRINTS == 0)
            printf("CUBE finished\n");

        if(ME == 0)
        {
          timer = RCCE_wtime() - timer;
          avg_time[1][i] +=timer;
          if(timer < min_time[1][i])
            min_time[1][i] = timer;
        }

        for(j=0;j<size;j++)
          if(buffer[j] != j%127)
          {
            printf("%d CUBE-ERROR in Position %d\n", ME, j);
            break;
          }

        for(j=0;j<size;j++)
          if(ME==0)
            buffer[j] = j%127;
          else
            buffer[j] = 'Z';

        RCCE_barrier(&RCCE_COMM_WORLD);

        if(ME == 0);
          timer = RCCE_wtime();

        RCCE_bcast(buffer, size,0,RCCE_COMM_WORLD);

        RCCE_barrier(&RCCE_COMM_WORLD);

        if(ME==0)
          if(x%PRINTS == 0)
          printf("RCCE finished\n");

      if(ME == 0)
      {
        timer = RCCE_wtime() - timer;
        avg_time[2][i] +=timer;
        if(timer < min_time[2][i])
          min_time[2][i] = timer;
      }

      for(j=0;j<size;j++)
        if(buffer[j] != j%127)
        {
          printf("%d RCCE_bcast-ERROR in Position %d\n", ME, j);
          break;
        }
  }

    avg_time[0][i] /=ROUNDS;
    avg_time[1][i] /=ROUNDS;
    avg_time[2][i] /=ROUNDS;

    free(buffer);

    if(ME==0)
    {
      if(WRITE_TO_FILE)
      {
        FILE *fp;

        char name[100];

        sprintf(name,"/shared/pichler/BENCHMARKS/DOPL/%d_cores/%d_%d_%d_%d_%d.txt",
            cores, cores, START_SIZE, STEP_SIZE, MAX_SIZE, ROUNDS);
        fp=fopen(name,"a");
        fprintf(fp, "%d\t%1.9lf\t%1.9lf\n", sizes[i], min_time[0][i], avg_time[0][i]);
        fclose(fp);

        sprintf(name,"/shared/pichler/BENCHMARKS/HYPERCUBE/%d_cores/%d_%d_%d_%d_%d.txt",
            cores, cores, START_SIZE, STEP_SIZE, MAX_SIZE, ROUNDS);
        fp=fopen(name,"a");
        fprintf(fp, "%d\t%1.9lf\t%1.9lf\n", sizes[i], min_time[1][i], avg_time[1][i]);
        fclose(fp);

        sprintf(name,"/shared/pichler/BENCHMARKS/RCCE_bcast/%d_cores/%d_%d_%d_%d_%d.txt",
            cores, cores, START_SIZE, STEP_SIZE, MAX_SIZE, ROUNDS);
        fp=fopen(name,"a");
        fprintf(fp, "%d\t%1.9lf\t%1.9lf\n", sizes[i], min_time[2][i], avg_time[2][i]);
        fclose(fp);
      }
    }
  }

  RCCE_barrier(&RCCE_COMM_WORLD);

  RCCE_finalize();
  return 0;
}
```

10.3 Hypercube

```
int calc_co(int p, int powerofq, int n)
{
  int co;

  if(p == 0)
    co = 0;
  else
  {
    if(p <= (n - powerofq))
      co = powerofq - 1 + p;
    else
    {
      if(p < powerofq)
        co = p;
      else
        co = p - powerofq + 1;
    }
  }
  return co;
}


int HYPERCUBE_bcast(char *buffer, int buffer_size)
{
  int YOU, ME, n, q, i, j, cur, tail = 0, rightmost = -1, head = 0, shift = 1, s, t, bit, jj;
  int u, co_i, rep, powerofq, counter, out, in, v, co, out_partner, partner, in_partner;
  int *dis, *switcharr;
  int parts, part_size = 4096, last_size;

  parts = buffer_size / part_size;

  if(parts * part_size != buffer_size)
  {
    parts++;
    last_size = buffer_size - (parts - 1) * part_size;
  }
  else
    last_size = part_size;


  ME = RCCE_ue();

  n = RCCE_num_ues();

  for (q = 0, powerofq = 1; powerofq < n; powerofq <<= 1)
    q++;

  if(powerofq == n)
  {
    dis = (int*)malloc(q * sizeof(int));

    if(ME == 0)
    {
      for(i = 0; i < q; i++)
        dis[i] = q;
    }
    else
    {
      for(head = 0, shift = 1; head < q; head++, shift<<=1)
      {
        if((ME & shift) == shift)
        {
          for(cur = tail; cur < head; cur++)
            dis[cur] = head - cur;

          tail = head;
          if(rightmost == -1)
            rightmost = head;
        }
      }
      for(cur = tail; cur < q; cur++)
        dis[cur] = q - cur + rightmost;
    }

    for(j = 0, shift = 1; j < parts + q - 1; j++, shift <<= 1)
    {
      jj = j % q;
      if(jj == 0)
        shift = 1;

      YOU = ME ^ shift;

      bit = ((ME & shift) == shift) ? 1 : 0;

      s = j - q + (1 - bit) * dis[jj];
      t = j - q + bit * dis[jj];

      if(s > parts - 1)
        s = parts - 1;

      if(t > parts - 1)
        t = parts - 1;

      if(ME < YOU)
      {
        if(s >= 0)
          if(s == parts - 1)
            RCCE_send(buffer + s * part_size, last_size, YOU);
          else
            RCCE_send(buffer + s * part_size, part_size, YOU);

        if(t >= 0 && ME > 0)
          if(t == parts - 1)
            RCCE_recv(buffer + t * part_size, last_size, YOU);
          else
            RCCE_recv(buffer + t * part_size, part_size, YOU);
      }
      else
      {
        if(t >= 0)
          if(t == parts - 1)
            RCCE_recv(buffer + t * part_size, last_size, YOU);
          else
            RCCE_recv(buffer + t * part_size, part_size, YOU);

        if(s >= 0 && YOU > 0)
          if(s == parts - 1)
            RCCE_send(buffer + s * part_size, last_size, YOU);
          else
            RCCE_send(buffer + s * part_size, part_size, YOU);
      }
    }
    free(dis);
  }
  else
  {
    powerofq /= 2;
    q--;

    dis = (int*)malloc(q * sizeof(int));
    switcharr = (int*)malloc(q * sizeof(int));

    co_i = calc_co(ME, powerofq, n);

    rep = (ME < powerofq) ? ME : co_i;

    if(ME == 0)
    {
      for(i = 0; i < q; i++)
        dis[i] = q;
    }
    else
    {
      for(head = 0, shift = 1; head < q; head++, shift <<= 1)
      {
        if((rep & shift) == shift)
        {
          for(cur = tail; cur < head; cur++)
            dis[cur] = head - cur;

          tail = head;
          if(rightmost == -1)
            rightmost = head;
        }
      }
      for(cur = tail; cur < q; cur++)
        dis[cur] = q - cur + rightmost;
    }

    counter = 0;
    for(i = 0, shift = 1; i < q; i++, shift <<= 1)
    {
      if((rep & shift) == shift)
        counter++;

      switcharr[i] = counter;
    }

    for(j = 0, shift = 1; j < parts + q - 1; j++, shift <<= 1)
    {
      jj = j % q;
      if(jj == 0)
        shift = 1;

      u = (j > 0) ? switcharr[q-1] * ((j - 1) / q) + switcharr[(j - 1) % q] : 0;
      co = calc_co(rep, powerofq, n);
      out = (1 - (u % 2)) * co + (u % 2) * rep;
      in = calc_co(out, powerofq, n);
      partner = rep ^ shift;
      v = (u + j / q) % 2;
      co = calc_co(partner, powerofq, n);
      out_partner = (1 - v) * co + v * partner;
      in_partner = calc_co(out_partner, powerofq, n);

      bit = ((rep & shift) == shift) ? 1 : 0;
      s = j - q + (1 - bit) * dis[jj];
      t = j - q + bit * dis[jj];

      if(s > parts - 1)
        s = parts - 1;

      if(t > parts - 1)
        t = parts - 1;

      if(ME == co_i)
      {
        if(ME < in_partner)
        {
          if(s >= 0)
            if(s == parts - 1)
```

```
       RCCE_send(buffer + s * part_size, last_size, in_partner);
      else
       RCCE_send(buffer + s * part_size, part_size, in_partner);

     if(t >= 0)
      if(t == parts - 1)
       RCCE_recv(buffer + t * part_size, last_size, out_partner);
      else
       RCCE_recv(buffer + t * part_size, part_size, out_partner);
    }
    else
    {
     if(t >= 0)
      if(t == parts - 1)
       RCCE_recv(buffer + t * part_size, last_size, out_partner);
      else
       RCCE_recv(buffer + t * part_size, part_size, out_partner);

     if(s >= 0)
      if(s == parts - 1)
       RCCE_send(buffer + s * part_size, last_size, in_partner);
      else
       RCCE_send(buffer + s * part_size, part_size, in_partner);
    }
   }
   else
   {
    if(ME == out)
    {
     if(s >= 0)
      if(s == parts - 1)
       RCCE_send(buffer + s * part_size, last_size, in_partner);
      else
       RCCE_send(buffer + s * part_size, part_size, in_partner);

     if(j - q - 1 >= 0)
      if(j - q - 1 == parts - 1)
       RCCE_recv(buffer + (j - q - 1) * part_size, last_size, in);
      else
       RCCE_recv(buffer + (j - q - 1) * part_size, part_size, in);
    }
    else
    {
     if(t >= 0)
      if(t == parts - 1)
       RCCE_recv(buffer + t * part_size, last_size, out_partner);
      else
       RCCE_recv(buffer + t * part_size, part_size, out_partner);

     if(j - q - 1 >= 0)
```

```
            if(j - q - 1 == parts - 1)
              RCCE_send(buffer + (j - q - 1) * part_size, last_size, out);
            else
              RCCE_send(buffer + (j - q - 1) * part_size, part_size, out);
        }
      }
    }

    if(ME != co_i)
    {
      u = switcharr[q - 1] * ((j - 1) / q) + switcharr[(j - 1) % q];
      co = calc_co(rep, powerofq, n);
      out = (1 - (u % 2)) * co + (u % 2) * rep;
      in = calc_co(out, powerofq, n);

      if(ME == out)
      {
        RCCE_recv(buffer + (parts - 2) * part_size, part_size, in);
        RCCE_send(buffer + (parts - 1) * part_size, last_size, in);
      }
      else
      {
        RCCE_send(buffer + (parts - 2) * part_size, part_size, out);
        RCCE_recv(buffer + (parts - 1) * part_size, last_size, out);
      }
    }

    free(dis);
    free(switcharr);
  }
  return 0;
}
```

10.4 DOPL

```c
#define min(x,y) ((x)<(y)?(x):(y))
#define equal(x,y) (((x[0])==(y[0])) && ((x[1])==(y[1])))

void assign(int *x, int *y)
{
  x[0] = y[0];
  x[1] = y[1];
}

inline int level(int* a, int* b, int imodd)
{
  int i;

  for(i = 0; i < 2 && a[(imodd + i) % 2] == b[(imodd + i) % 2]; i++);

  return i;
}

void pipe(char *buffer_r, int r, int kr, char *buffer_s, int s, int ks,
    int kalt, int imodd, int* me, int r_buffer_size, int s_buffer_size,
    int*m, int **processor_map, iRCCE_WAIT_LIST *general_waitlist,
    iRCCE_RECV_REQUEST *recv_requests, iRCCE_SEND_REQUEST *send_requests);

void dopl(char *buffer, int buffer_size)
{
  int rows, i, j, ME, kalt, d=2, r, kr, imodd, s, ks;
  int kalt_size = 8192, last_size, r_buffer_size, s_buffer_size;
  int me[2], head[2], src[2] = {0,0}, tail[2], m[2], k[2] = {4,4};

  iRCCE_WAIT_LIST general_waitlist;
  iRCCE_RECV_REQUEST* recv_requests = NULL;
  iRCCE_SEND_REQUEST* send_requests = NULL;

  rows = RCCE_num_ues() / 6;

  int **processor_map = (int**)malloc(rows * sizeof(int*));
  recv_requests = (iRCCE_RECV_REQUEST*)malloc(sizeof(iRCCE_RECV_REQUEST) * 4);
  send_requests = (iRCCE_SEND_REQUEST*)malloc(sizeof(iRCCE_SEND_REQUEST) * 4);

  m[0] = rows;
  m[1] = 6;

  for(i = 0; i < rows; i++)
    processor_map[i] = (int*)malloc(6 * sizeof(int));

  for(i = 0; i < rows ; i++)
    for(j = 0; j < 6; j++)
      if(i % 2 == 0)
        processor_map[i][j] = (i / 2) * 12 + 2 * j;
      else
        processor_map[i][j] = (i / 2) * 12 + 2 * j + 1;
```

```

ME = RCCE_ue();
iRCCE_init_wait_list(&general_waitlist);

for(i = 0; i < rows; i++)
  for(j = 0; j < 6; j++)
    if(processor_map[i][j] == ME)
    {
      me[0] = i;
      me[1] = j;
    }

kalt = buffer_size / kalt_size;

if(kalt_size * kalt != buffer_size)
{
  kalt++;
  last_size = buffer_size - kalt_size * (kalt - 1);
}
else
  last_size = kalt_size;

for(i = 0; i < kalt + d - 1; i++)
{
  imodd = i % d;
  assign(head, me);
  head[imodd] = src[imodd];
  assign(tail, head);

  tail[imodd] = (head[imodd] + m[imodd] - 1) % m[imodd];

  if(equal(me, head))
    if(equal(me, src))
      r = -1;
    else
    {
      r = i - d;
      kr = 1;
    }
  else
  {
    r = min(kalt - 1, i + level(head, src, imodd) - d);
    kr = k[imodd];
  }

  if(equal(me, tail))
    if(equal(head, src))
      s = -1;
    else
    {
        s = i - d;
        ks = 1;
      }
    else
    {
      s = min(kalt - 1, i + level(head, src, imodd) - d);
      ks = k[imodd];
    }

    r_buffer_size = (r == kalt - 1) ? last_size : kalt_size;
    s_buffer_size = (s == kalt - 1) ? last_size : kalt_size;

    if(r >= 0 && s >= 0)
      pipe(buffer + kalt_size * r, r, kr, buffer + kalt_size * s, s,
      ks, kalt, imodd, me, r_buffer_size, s_buffer_size, m, processor_map,
      &general_waitlist, recv_requests, send_requests);

    else if(r >= 0)
      pipe(buffer + kalt_size * r, r, kr, NULL, 0, 0, kalt, imodd, me,
      r_buffer_size, 0, m, processor_map, &general_waitlist, recv_requests, send_requests);

    else if(s >= 0)
      pipe(NULL, 0, 0, buffer + kalt_size * s, s, ks, kalt, imodd, me,
      0, s_buffer_size, m, processor_map, &general_waitlist, recv_requests, send_requests);
  }

  for(i = 0; i < rows; i++)
    free(processor_map[i]);

  free(processor_map);

  free(recv_requests);
  free(send_requests);
}

void pipe(char *buffer_r, int r, int kr, char *buffer_s, int s, int ks, int kalt,
    int imodd, int* me, int r_buffer_size, int s_buffer_size, int*m, int **processor_map,
    iRCCE_WAIT_LIST *general_waitlist, iRCCE_RECV_REQUEST *recv_requests,
    iRCCE_SEND_REQUEST *send_requests)
{
  int prev[2], next[2];
  int i, kr_last_buffer_size = 0, ks_last_buffer_size = 0;


  const int pipe_size = 2048;

  assign(prev, me);
  prev[imodd]=(prev[imodd] + m[imodd] - 1) % m[imodd];

  next[imodd] = (next[imodd] + 1) % m[imodd];

  if(r>=0 && kr && r_buffer_size > 0)
  {
    if(kr == 4)
    {
      kr = r_buffer_size / pipe_size;

      if(kr * pipe_size != r_buffer_size)
      {
        kr++;
        kr_last_buffer_size = r_buffer_size - (kr - 1) * pipe_size;
      }
      else
        kr_last_buffer_size = pipe_size;
    }
    else
      kr_last_buffer_size = r_buffer_size;
  }

  if(s >= 0 && ks && s_buffer_size > 0)
  {
    if(ks == 4)
    {
      ks = s_buffer_size / pipe_size;

      if(ks * pipe_size != s_buffer_size)
      {
        ks++;
        ks_last_buffer_size = s_buffer_size - (ks - 1) * pipe_size;
      }
      else
        ks_last_buffer_size = pipe_size;
    }
    else
      ks_last_buffer_size = s_buffer_size;
  }

  if(r >= 0 && kr > 0 && r_buffer_size > 0)
  {
    for(i = 0; i < kr - 1; i++)
    {
      iRCCE_irecv(buffer_r + i * pipe_size, pipe_size,
          processor_map[prev[0]][prev[1]], recv_requests + i);
      iRCCE_add_to_wait_list(general_waitlist, NULL, recv_requests + i);
    }

    iRCCE_irecv(buffer_r + (kr - 1) * pipe_size, kr_last_buffer_size,
        processor_map[prev[0]][prev[1]], recv_requests + kr - 1);
    iRCCE_add_to_wait_list(general_waitlist, NULL, recv_requests + kr - 1);
  }

  if(buffer_r == buffer_s)
  {
    if(s>=0 && ks>0 && s_buffer_size > 0)
    {
      for(i = 0; i < ks - 1; i++)
      {
        iRCCE_irecv_wait(recv_requests + i);
        iRCCE_isend(buffer_s + i * pipe_size, pipe_size, processor_map[next[0]][next[1]],
            send_requests + i);
        iRCCE_add_to_wait_list(general_waitlist, send_requests + i, NULL);
      }

       iRCCE_irecv_wait(recv_requests + ks - 1);
       iRCCE_isend(buffer_s + (ks - 1) * pipe_size, ks_last_buffer_size,
           processor_map[next[0]][next[1]], send_requests + ks - 1);
       iRCCE_add_to_wait_list(general_waitlist, send_requests + ks - 1, NULL);
    }
  }
  else
  {
    if(s >= 0 && ks > 0 && s_buffer_size > 0)
    {
      for(i = 0; i < ks - 1; i++)
      {
        iRCCE_isend(buffer_s + i * pipe_size, pipe_size, processor_map[next[0]][next[1]],
            send_requests + i);
        iRCCE_add_to_wait_list(general_waitlist, send_requests + i, NULL);
      }

       iRCCE_isend(buffer_s + (ks - 1) * pipe_size, ks_last_buffer_size,
           processor_map[next[0]][next[1]], send_requests + ks - 1);
       iRCCE_add_to_wait_list(general_waitlist, send_requests + ks - 1, NULL);
    }
  }

  iRCCE_wait_all(general_waitlist);
}
```

10.5 Hypercube Best Size Benchmark Main

```
#include <string.h>
#include <stdio.h>
#include "RCCE.h"
#include <math.h>
#include "iRCCE.h"
#include "HYPERCUBE.h"

#define PARTS 1
#define PART_SIZE 1900000
#define ROUNDS 5


int RCCE_APP(int argc, char **argv)
{
  int ME, i, j;
  int x;
  double timer = 0.0, avg_time;
  double min_time = 9999;
  char *buffer = (char*)malloc(PARTS*PART_SIZE*sizeof(char));

  RCCE_init(&argc, &argv);
  iRCCE_init();

  ME = RCCE_ue();

  for(i=0;i<PARTS*PART_SIZE;i++)
    if(ME == 0)
      buffer[i] = 'A';
    else
      buffer[i] = 'Z';

  RCCE_barrier(&RCCE_COMM_WORLD);
  for(i=1;i<=2048;i*=2)
  {
    min_time = 9999;
    avg_time = 0.0;

    for(x=0;x<ROUNDS;x++)
    {
      RCCE_barrier(&RCCE_COMM_WORLD);

      if(ME == 0);
        timer = RCCE_wtime();

      HYPERCUBE_bcast(buffer, PARTS* PART_SIZE, i);

      if(ME == 0)
      {
        timer = RCCE_wtime() - timer;
        avg_time +=timer;
        if(timer < min_time)
          min_time = timer;
      }

      for(j=0;j<PARTS*PART_SIZE;j++)
        if(buffer[j] != 'A')
        {
          printf("%d ERROR in Position %d\n", ME, j);
          break;
        }

      for(j=0;j<PARTS*PART_SIZE;j++)
        if(ME == 0)
          buffer[j] = 'A';
        else
          buffer[j] = 'Z';
    }
    if(ME==0)
    {
      avg_time /= ROUNDS;
      printf("PARTSIZE: %d, min_time: %.15f avg_time: %.15f\n",524288/i,min_time,avg_time);
    }
  }
  RCCE_barrier(&RCCE_COMM_WORLD);

  free(buffer);
  RCCE_finalize();
}
```

10.6 Hypercube Best Size Benchmark

```
int calc_co(int p, int powerofq, int n)
{
  int co;

  if(p==0)
    co=0;
  else
  {
    if(p<=(n-powerofq))
      co=powerofq-1+p;
    else
    {
      if(p<powerofq)
        co=p;
      else
        co=p-powerofq+1;
    }
  }
  return co;
}


int HYPERCUBE_bcast(char *buffer, int buffer_size, int divisor)
{
  int YOU, ME, n, q, i, j, cur, tail = 0, rightmost = -1, head = 0, shift=1, s, t, bit, jj;
  int u, co_i, rep, powerofq, counter, out, in, v, co, out_partner, partner, in_partner;
  int *dis, *switcharr;
  int parts, part_size = 524288/divisor, last_size;

  parts = buffer_size/part_size;

  if(parts*part_size != buffer_size)
  {
    parts++;
    last_size = buffer_size - (parts-1)*part_size;
  }
  else
    last_size = part_size;


  ME = RCCE_ue();

  n = RCCE_num_ues();

  for (q=0,powerofq=1; powerofq<n; powerofq<<=1)
    q++;

  if(powerofq == n)
  {
    dis = (int*)malloc(q*sizeof(int));

    if(ME == 0)
    {
      for(i=0;i<q;i++)
        dis[i]=q;
    }
    else
    {
      for(head = 0, shift = 1; head < q; head++, shift<<=1)
      {
        if((ME & shift) == shift)
        {
          for(cur = tail; cur < head; cur++)
            dis[cur] = head-cur;

          tail = head;
          if(rightmost == -1)
            rightmost = head;
        }
      }
      for(cur = tail; cur < q; cur++)
        dis[cur] = q - cur + rightmost;
    }

    for(j = 0, shift = 1; j < parts + q - 1; j++, shift<<=1)
    {
      jj = j%q;
      if(jj == 0)
        shift = 1;

      YOU = ME^shift;

      bit = ((ME & shift) == shift)? 1 : 0;

      s = j - q + (1 - bit) * dis[jj];
      t = j - q + bit * dis[jj];

      if(s>parts-1)
        s = parts-1;

      if(t>parts-1)
        t = parts-1;

      if(ME<YOU)
      {
        if(s>=0)
          if(s==parts-1)
            RCCE_send(buffer+s*part_size, last_size, YOU);
          else
            RCCE_send(buffer+s*part_size, part_size, YOU);

        if(t>=0 && ME>0)
          if(t==parts-1)
            RCCE_recv(buffer+t*part_size, last_size, YOU);
          else
            RCCE_recv(buffer+t*part_size, part_size, YOU);
      }
      else
      {
        if(t>=0)
          if(t==parts-1)
            RCCE_recv(buffer+t*part_size, last_size, YOU);
          else
            RCCE_recv(buffer+t*part_size, part_size, YOU);

        if(s>=0 && YOU>0)
          if(s==parts-1)
            RCCE_send(buffer+s*part_size, last_size, YOU);
          else
            RCCE_send(buffer+s*part_size, part_size, YOU);
      }
    }
    free(dis);
  }
  else
  {
    powerofq /= 2;
    q--;

    dis = (int*)malloc(q*sizeof(int));
    switcharr = (int*)malloc(q*sizeof(int));

    co_i = calc_co(ME, powerofq, n);

    rep=(ME<powerofq)? ME : co_i;

    if(ME == 0)
    {
      for(i=0;i<q;i++)
        dis[i]=q;
    }
    else
    {
      for(head = 0, shift = 1; head < q; head++, shift<<=1)
      {
        if((rep & shift) == shift)
        {
          for(cur = tail; cur < head; cur++)
            dis[cur] = head-cur;

          tail = head;
          if(rightmost == -1)
            rightmost = head;
        }
      }
      for(cur = tail; cur < q; cur++)
        dis[cur] = q - cur + rightmost;
    }

    counter = 0;
    for(i=0,shift=1;i<q;i++,shift<<=1)
    {
      if((rep&shift)==shift)
        counter++;

      switcharr[i]=counter;
    }

    for(j = 0, shift = 1; j < parts + q - 1; j++, shift<<=1)
    {
      jj = j%q;
      if(jj == 0)
        shift = 1;

      u = (j>0) ? switcharr[q-1]*((j-1)/q) + switcharr[(j-1)%q] : 0;
      co = calc_co(rep, powerofq, n);
      out = (1-(u%2))*co+(u%2)*rep;
      in = calc_co(out, powerofq, n);
      partner = rep^shift;
      v = (u+j/q)%2;
      co = calc_co(partner, powerofq, n);
      out_partner = (1-v)*co+v*partner;
      in_partner = calc_co(out_partner,powerofq,n);

      bit = ((rep & shift) == shift)? 1 : 0;
      s = j - q + (1 - bit) * dis[jj];
      t = j - q + bit * dis[jj];

      if(s>parts-1)
        s = parts-1;

      if(t>parts-1)
        t = parts-1;

      if(ME == co_i)
      {
        if(ME < in_partner)
        {
          if(s>=0)
            if(s==parts-1)
          RCCE_send(buffer+s*part_size, last_size, in_partner);
         else
          RCCE_send(buffer+s*part_size, part_size, in_partner);

        if(t>=0)
         if(t==parts-1)
          RCCE_recv(buffer+t*part_size, last_size, out_partner);
         else
          RCCE_recv(buffer+t*part_size, part_size, out_partner);
       }
       else
       {
        if(t>=0)
         if(t==parts-1)
          RCCE_recv(buffer+t*part_size, last_size, out_partner);
         else
          RCCE_recv(buffer+t*part_size, part_size, out_partner);

        if(s>=0)
         if(s==parts-1)
          RCCE_send(buffer+s*part_size, last_size, in_partner);
         else
          RCCE_send(buffer+s*part_size, part_size, in_partner);
       }
      }
      else
      {
       if(ME==out)
       {
        if(s>=0)
         if(s==parts-1)
          RCCE_send(buffer+s*part_size, last_size, in_partner);
         else
          RCCE_send(buffer+s*part_size, part_size, in_partner);

        if(j-q-1 >= 0)
         if(j-q-1 == parts-1)
          RCCE_recv(buffer+(j-q-1)*part_size, last_size, in);
         else
          RCCE_recv(buffer+(j-q-1)*part_size, part_size, in);
       }
       else
       {
        if(t>=0)
         if(t==parts-1)
          RCCE_recv(buffer+t*part_size, last_size, out_partner);
         else
          RCCE_recv(buffer+t*part_size, part_size, out_partner);

        if(j-q-1 >= 0)
```

```
                if(j-q-1 == parts-1)
                  RCCE_send(buffer+(j-q-1)*part_size, last_size, out);
                else
                  RCCE_send(buffer+(j-q-1)*part_size, part_size, out);
            }
          }
        }

        if(ME!=co_i)
        {
          u = switcharr[q-1]*((j-1)/q) + switcharr[(j-1)%q];
          co = calc_co(rep, powerofq, n);
          out = (1-(u%2))*co+(u%2)*rep;
          in = calc_co(out, powerofq, n);

          if(ME==out)
          {
            RCCE_recv(buffer+(parts-2)*part_size, part_size, in);
            RCCE_send(buffer+(parts-1)*part_size, last_size, in);
          }
          else
          {
            RCCE_send(buffer+(parts-2)*part_size, part_size, out);
            RCCE_recv(buffer+(parts-1)*part_size, last_size, out);
          }
        }

        free(dis);
        free(switcharr);
      }
      return(0);
    }
```

10.7 DOPL Size Benchmark

```c
#include <string.h>
#include <stdio.h>
#include "RCCE.h"
#include <math.h>
#include "iRCCE.h"

#define PARTS 1
#define PART_SIZE 50000
#define ROUNDS 15


#define min(x,y) ((x)<(y)?(x):(y))
#define equal(x,y) (((x[0])==(y[0])) && ((x[1])==(y[1])))

void assign(int *x, int *y)
{
        x[0]=y[0];
        x[1]=y[1];
}

inline int level(int* a, int* b, int imodd)
{
  int i;

  for(i=0;i<2 && a[(imodd + i)%2]==b[(imodd + i)%2];i++);

  return i;
}

void dopl(char *buffer, int buffer_size, int *k);

void pipe(char *buffer_r, int r, int kr, char *buffer_s, int s,
    int ks, int kalt, int imodd, int* me, int r_buffer_size,
    int s_buffer_size, int*m, int **processor_map,
    iRCCE_WAIT_LIST *general_waitlist, int ME);



int RCCE_APP(int argc, char **argv)
{
  int ME, i, j;
  int k[2] = {1,1}, x;
  double timer = 0.0;
  double min_time = 9999;
  char *buffer = (char*)malloc(PART_SIZE*sizeof(char));

  RCCE_init(&argc, &argv);
  iRCCE_init();

  ME = RCCE_ue();

  for(i=0;i<PART_SIZE;i++)
    if(ME == 0)
      buffer[i] = 'A';
    else
      buffer[i] = 'Z';

  RCCE_barrier(&RCCE_COMM_WORLD);

  for(i=1;i<=32;i++)
  {
    k[0]=i;
    k[1]=i;

    min_time = 9999;

    for(x=0;x<ROUNDS;x++)
    {
      RCCE_barrier(&RCCE_COMM_WORLD);

      if(ME == 0);
        timer = RCCE_wtime();

      dopl(buffer, PART_SIZE, k);

      RCCE_barrier(&RCCE_COMM_WORLD);

      if(ME == 0)
      {
        timer = RCCE_wtime() - timer;

        if(timer < min_time)
          min_time = timer;
      }

      for(j=0;j<PART_SIZE;j++)
        if(buffer[j] != 'A')
        {
          printf("%d ERROR in Position %d\n", ME, j);
          break;
        }

      for(j=0;j<PART_SIZE;j++)
        if(ME == 0)
          buffer[j] = 'A';
        else
          buffer[j] = 'Z';
    }
    if(ME==0)
      printf("Pipesize: %d, time: %.15f\n",8192/k[0],min_time);
```

```c
    RCCE_barrier(&RCCE_COMM_WORLD);
  }
  free(buffer);
  RCCE_finalize();
}

void dopl(char *buffer, int buffer_size, int *k)
{
  int rows, i, j, ME, kalt, d=2, r, kr, imodd, s, ks, kalt_size = 8192;
  int last_size, r_buffer_size, s_buffer_size;
  int me[2], head[2], src[2] = {0,0}, tail[2], m[2];
  iRCCE_WAIT_LIST general_waitlist;
  rows = RCCE_num_ues()/6;
  int **processor_map = (int**)malloc(rows*sizeof(int*));

  m[0] = rows;
  m[1] = 6;

  for(i=0;i<rows;i++)
    processor_map[i] = (int*)malloc(6*sizeof(int));

  for(i=0;i<rows;i++)
    for(j=0;j<6;j++)
      if(i%2==0)
        processor_map[i][j] = (i/2)*12+2*j;
      else
        processor_map[i][j] = (i/2)*12+2*j+1;

  ME = RCCE_ue();
  iRCCE_init_wait_list(&general_waitlist);

  for(i=0;i<rows;i++)
    for(j=0;j<6;j++)
      if(processor_map[i][j] == ME)
      {
        me[0] = i;
        me[1] = j;
      }

  kalt = buffer_size/kalt_size;

  if(kalt_size * kalt != buffer_size)
  {
    kalt++;
    last_size = buffer_size-kalt_size*(kalt-1);
  }
  else
    last_size = kalt_size;

  for(i=0;i<kalt+d-1;i++)
```

```
{
  imodd = i%d;
  assign(head,me);
  head[imodd]=src[imodd];
  assign(tail,head);

  tail[imodd] = (head[imodd] + m[imodd] - 1) % m[imodd];

  if(equal(me,head))  //if me == head
    if(equal(me,src))  //if me == src
      r = -1;
    else
    {
      r = i-d;
      kr = 1;
    }
  else
  {
    r = min(kalt-1, i + level(head, src, imodd)-d);
    kr = k[imodd];
  }

  if(equal(me,tail)) //if me == tail
    if(equal(head,src)) //if head == src
      s = -1;
    else
    {
      s = i-d;
      ks = 1;
    }
  else
  {
    s = min(kalt-1, i+level(head, src, imodd)-d);
    ks = k[imodd];
  }

  r_buffer_size = (r==kalt-1) ? last_size : kalt_size;
  s_buffer_size = (s==kalt-1) ? last_size : kalt_size;

  if(r>=0 && s>=0)
    pipe(buffer+kalt_size*r, r, kr, buffer+kalt_size*s, s, ks, kalt, imodd, me,
      r_buffer_size, s_buffer_size,m,processor_map, &general_waitlist, ME);

  else if(r>=0)
    pipe(buffer+kalt_size*r, r, kr, NULL, 0, 0, kalt, imodd, me, r_buffer_size,
    0, m,processor_map, &general_waitlist, ME);

  else if(s>=0)
    pipe(NULL, 0, 0, buffer+kalt_size*s, s, ks, kalt, imodd, me, 0, s_buffer_size,
    m,processor_map, &general_waitlist, ME);
  }

  for(i=0;i<rows;i++)
          free(processor_map[i]);

  free(processor_map);
}

void pipe(char *buffer_r, int r, int kr, char *buffer_s, int s, int ks, int kalt, int imodd,
    int* me, int r_buffer_size, int s_buffer_size, int*m, int **processor_map,
    iRCCE_WAIT_LIST *general_waitlist, int ME)
{
  int prev[2], next[2];
  int i, kr_buffer_size = 0, ks_buffer_size = 0, kr_last_buffer_size = 0;
  int ks_last_buffer_size = 0;
  iRCCE_RECV_REQUEST* recv_requests = NULL;
  iRCCE_SEND_REQUEST* send_requests = NULL;

  assign(prev,me);
  prev[imodd]=(prev[imodd]+m[imodd]-1)%m[imodd];

  assign(next,me);
  next[imodd]=(next[imodd] + 1)%m[imodd];

  if(r>=0 && kr && r_buffer_size > 0)
  {
    if(r_buffer_size/kr*kr==r_buffer_size)
      kr_buffer_size = kr_last_buffer_size = r_buffer_size/kr;
    else
    {
      kr_buffer_size = r_buffer_size/kr + 1;
      if(r_buffer_size/kr_buffer_size * kr_buffer_size != r_buffer_size)
        kr = r_buffer_size/kr_buffer_size+1;

      kr_last_buffer_size = r_buffer_size - (kr-1)*kr_buffer_size;
    }

    recv_requests = (iRCCE_RECV_REQUEST*)malloc(sizeof(iRCCE_RECV_REQUEST)*kr);
  }

  if(s>=0 && ks && s_buffer_size > 0)
  {
    if(s_buffer_size/ks*ks == s_buffer_size)
      ks_buffer_size = ks_last_buffer_size = s_buffer_size/ks;
    else
    {
      ks_buffer_size = s_buffer_size/ks+1;
      if(s_buffer_size/ks_buffer_size * ks_buffer_size != s_buffer_size)
        ks = s_buffer_size/ks_buffer_size+1;

      ks_last_buffer_size = s_buffer_size - (ks-1) * ks_buffer_size;
    }

    send_requests = (iRCCE_SEND_REQUEST*)malloc(sizeof(iRCCE_SEND_REQUEST)*ks);
  }

  if(r>=0 && kr>0 && r_buffer_size > 0)
  {
    for(i=0;i<kr-1;i++)
    {
      iRCCE_irecv(buffer_r+i*kr_buffer_size, kr_buffer_size,
          processor_map[prev[0]][prev[1]], recv_requests+i);
      iRCCE_add_to_wait_list(general_waitlist, NULL, recv_requests+i);
    }
    if(kr_last_buffer_size > 0)
    {
      iRCCE_irecv(buffer_r+(kr-1)*kr_buffer_size, kr_last_buffer_size,
          processor_map[prev[0]][prev[1]], recv_requests+kr-1);
      iRCCE_add_to_wait_list(general_waitlist, NULL, recv_requests+kr-1);
    }
  }

  if(buffer_r == buffer_s)
  {
    if(s>=0 && ks>0 && s_buffer_size > 0)
    {
      for(i=0;i<ks-1;i++)
      {
        iRCCE_irecv_wait(recv_requests +i);
        iRCCE_isend(buffer_s+i*ks_buffer_size, ks_buffer_size,
            processor_map[next[0]][next[1]], send_requests + i);
        iRCCE_add_to_wait_list(general_waitlist, send_requests+i, NULL);
      }
      if(ks_last_buffer_size > 0)
      {
        iRCCE_irecv_wait(recv_requests + ks -1);
        iRCCE_isend(buffer_s+(ks-1)*ks_buffer_size, ks_last_buffer_size,
            processor_map[next[0]][next[1]], send_requests + ks -1);
        iRCCE_add_to_wait_list(general_waitlist, send_requests+ks-1, NULL);
      }
    }
  }
  else
  {
    if(s>=0 && ks > 0 && s_buffer_size > 0)
    {
      for(i=0;i<ks-1;i++)
      {
        iRCCE_isend(buffer_s+i*ks_buffer_size, ks_buffer_size,
            processor_map[next[0]][next[1]], send_requests + i);
            iRCCE_add_to_wait_list(general_waitlist, send_requests+i, NULL);
        }
        if(ks_last_buffer_size > 0)
        {
            iRCCE_isend(buffer_s+(ks-1)*ks_buffer_size, ks_last_buffer_size,
                processor_map[next[0]][next[1]], send_requests + ks -1);
            iRCCE_add_to_wait_list(general_waitlist, send_requests+ks-1, NULL);
        }
      }
    }

    iRCCE_wait_all(general_waitlist);

    free(recv_requests);
    free(send_requests);
}
```