FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

PhD thesis

Morten Nørgaard Larsen

Parallel Libraries to support High-Level Programming

Academic advisor: Brian Vinter

Submitted: 14/03/2013

The greatest challenge to any thinker
is stating the problem in a way
that will allow a solution

– Bertrand Russell (1872-1970)

Abstract / Resumé

Abstract

The development of computer architectures during the last ten years have forced programmers to move towards writing parallel programs instead of sequential ones. The homogenous multi-core architectures from the major CPU producers like Intel and AMD has led this trend, but the introduction of the more exotic though short-lived heterogeneous CELL Broadband Engine (CELL-BE) architecture added to this shift. Furthermore, the use of cluster computers made of commodity hardware and specialized supercomputers have greatly increased in both industry as well as in the academic world. Finally, the general increase in the usage of graphic cards for general-purpose programming (GPGPU) have meant that programmers today must be able to write parallel programs that cannot only utilize small number computational cores but perhaps hundreds or even thousands. However, most programmers will agree that doing so is not a simple task and for many non-computer scientists, like chemists and physicists writing programs for simulating their experiments, the task can easily become overwhelming.

During the last decades, a lot of research efforts have been put into how to create tools that will simplify writing parallel programs by raising the abstraction level, so that the programmers can focus on implementing their algorithms and not on the details of the underlying hardware. The outcome has ranged from ideas on of automating the parallelization of sequential programs to presenting a cluster of machines as they would be a single machine. In between is a number of tools helping the programmers handle communication, share data, run loops in parallel, handle algorithms mining huge amounts of data etc. Even though most of them do a good job performance-wise, almost all of them require that the programmers learn a new programming language or at least forces them to learn new methods and/or ways of writing code.

For the first part, this thesis will focus on simplifying the task of writing parallel programs for programmers, but especially for the large group of non-computer scientists. I will start by presenting an extension based on Communicating Sequential Processes (CSP) for a distributed shared memory system for the CELL Broadband Engine. This extension consists of a channel model and of a thread library for the CELL's specialized computational units enabling them to run multiple (CSP) processes. Overall, the CSP model requires the programmer to think a bit differently, but at the same time the implemented algorithms will perform very well, as shown by the initial tests presented.

In the second part of this thesis, I will change focus from the CELL-BE architecture to the more traditionally x86 architecture and the Microsoft .NET framework. Normally, one would not directly think of the .NET framework when talking scientific applications, but Microsoft has in the last couple of versions of .NET introduce a number of tools for writing parallel and high performance code. The first section examines how programmers can run parts of a program like a loop in parallel without directly programming the underlying hardware. The presented tool will be able to run the body of the method in parallel including handling the consistency of any shared data which the programmer accesses within the loop body. Doing so includes implementing a distributed shared memory system along with the MESI protocol on top of the .NET framework. However, during the implementation and while testing, it became clear that the lack of information regarding what shared data a method accesses greatly limits the overall performance. Moreover, the overhead of building a distributed shared memory system along with a consistency model on top of .NET became too large. Therefore, the work is repeated in another approach, which will force programmers to define what data a method will access when executed. Inspired by CSP, I define a set of rules dictating how programmers should write a method including input parameters, output values, and accesses of shared data. These rules make it possible to get more information, which in turns allows us to build a new tool system that does not need a distributed shared memory system or a consistency model. However, programmers can still invoke methods, and the tool will transparently run the method in parallel on a platform consisting of workstations, servers, and cloud instances. Overall, this increases the effort required by the programmers but greatly improves performance, as the initial tests shows.

Resumé

Udviklingen af computer arkitekturer har gennem de seneste 10 år gjort, at programmører i dag skal kunne skrive parallelle programmer i stedet for sekventielle programmer. Årsagen hertil er at finde i udviklingen af homogene multikerne arkitekturer fra store producenter såsom Intel og AMD, men også udviklingen af den anderledes om end kortlivede heterogene CELL Broadband Engine (CELL-BE) arkitektur fra IBM. Derudover er antallet af klynge computere samlet af almindelige computere samt antallet af supercomputere steget både i virksomheder men også i den akademiske verden. Endelig, har brugen af grafikkort til almindelig programmeringsformål betydet at programmører i dag må være i stand til at skrive parallelle programmer der ikke kun kan udnytte få, men måske hundrede eller tusinde beregningsenheder. De fleste programmører ved at denne opgave ikke er simpel og for gruppen af forskere såsom kemikere og fysikere, der skriver programmer for at simulere forsøg, kan opgaven være uoverkommelig.

Igennem de sidste årtier har datalogoer forsket i redskaber til at simplificere det at skrive parallelle programmer ved at løfte abstraktionsniveauet, således at programmørerne kan fokuserer på at implementere deres algoritmer og ikke på at skulle forstå den underliggende hardware m.v. Resultatet heraf har været idéer lige fra automatisk parallelisering af sekventielle programmer til klynge computere hvilke fra programmørens synspunkt ligner en helt almindelig computer. Imellem disse findes bl.a. en række redskaber som kan hjælpe programmørerne til at håndtere kommunikation, dele data, køre loop parallelt samt algoritmer til at udvinde information fra store data mængder. Selvom de fleste af redskaber giver en fornuftig ydelse, tvinger de fleste af dem programmøerne til at lære et nyt programmeringssprog eller til at lære nye metoder at skrive deres programmer på.

Den første del af denne afhandling fokuserer på at gøre det simplere for programmører, men specielt for den store gruppe af forskere uden datalogisk baggrund, at skrive parallelle programmer. Først præsenteres en udvidelse, til et distribueret delt hukommelses system, baseret på Communicating Sequential Processes (CSP). Denne udvidelse består af en kanal model samt et tråd bibliotek, til CELL'ens specialiserede beregningsenheder, hvilket gør det muligt for dem at køre flere samtidige CSP processer på hver beregningsenhed. CSP modellen gør at programmøren skal tænke en smule anderledes, men samtidigt viser de indledende test af algoritmer kan opnå god ydelse.

I den anden del af afhandlingen flyttes fokus fra CELL-BE arkitekturen til den mere traditionelle x86 arkitektur, og Microsoft .NET. Normalt anses .NET ikke som et redskab til at skrive videnskabelige programmer med, men Microsoft har i de seneste versioner introduceret en række redskaber til at skrive parallelle algoritmer og højt ydende kode. Først undersøges, hvordan programmører kan køre dele af deres programmer, f.eks. et loop, parallelt uden at tage hensyn til den bagvedliggende hardware. Det udviklede redskab kan eksekvere en metode parallelt, samtidigt med at der holdes styr på konsistensen af de delte data som metoden tilgår. Redskabet inkluder et distribueret delt hukommelses system samt MESI protokollen, begge implementeret ovenpå .NET. Under implementeringen og afprøvningsfasen blev det klart, at den manglende information om hvordan delt data skal tilgås, i stor grad miniminerede den overordnede ydelse. Ydermere, var meromkostningerne ved at bygge et distribueret delt hukommelses system samt en konsistens model oven på .NET for store. Derfor blev det gjort endnu et forsøg, hvori programmøren tvinges til at definere hvordan metoder tilgår delt data. Inspireret af CSP modellen, opsættes krav til en metodes input parameter, output og tilgange til delt data. Disse regler gør det muligt at få mere information, hvilket medfører at man kan bygge et lignende system som ikke indeholder en distribueret delt hukommelses eller konsistens model. Programmøren kan eksekverer sådanne metoder og redskabet vil transparent køre metoden parallelt, på en platform bestående af arbejdsstationer, serverer samt sky instanser. Overordnet, hæver redskabet kravene til programmøren, men samtidigt viser de indledende tests at ydelsen kan forbedres markant.

Acknowledgements

As a partial fulfillment of the requirements of obtaining the academic degree of Doctor of Philosophy (PhD) this thesis was on the 14th of March 2013 submitted to the Faculty of Science, University of Copenhagen. The thesis describes the results obtained during my PhD study. I carried out the work at the University of Copenhagen from March 2010 to March 2013 with a two month stay at Lehrstuhl für Parallele und Verteilte Systeme, Ruprecht-Karls-Universität Heidelberg in Heidelberg, Germany from November 2011 through December 2011. The Innovation Consortium supported this research with grant 09-052139.

Foremost, I would like to thank my supervisor Professor Brian Vinter for his help and enthusiastic guidance. Thank you for being able to introduce the bright side of a problem, for giving me the opportunity to work in a well structured group, and for giving me the freedom to work from my temporary home in Heidelberg through out 2012.

I am grateful to Professor Dr. Artur Andrzejak for the opportunity to stay in his group in Heidelberg, but especially to Dr. Sascha Hunold for having the time for many good discussions and giving me some advise on the life as a PhD student.

I would also like to thank my fellow students and co-workers at the University of Copenhagen for giving me an enjoyable place to work: Jonas Bardino, Troels Blum, Rune Friborg, Christian Jacobsen, Simon Lund, Martin Rehr, Benjamin Sedoc, Jesper Rude Selknæs, Kenneth Skovhede, and Yan Wang.

A special thanks to Kenneth Skovhede for having been there with me from the start – through the first courses, the final exams, and for always having the time to discuss this or that problem. It has been a great pleasure to work with you and hopefully our paths will cross many times in the future both work-wise and privately.

Last but not least, thanks to my friends, family and especially to my fiancée Mie. Thank you for your love and support during the last 10+ years

Morten Nørgaard Larsen
Copenhagen – 14.03.2013

Contents

Introduction to the Thesis

Over the last decades developing scientific applications for data analysis, modelling, simulation, and visualization has changed greatly from writing sequential code to writing parallel code, which runs on systems of tens, hundreds or even thousands of computers. The reason for this change is that the architecture of computers shifted from using single-core to multi-cores CPUs. In the nineties most scientific applications software ran on single core machines. When a scientist, e.g. a chemist, had written a program for calculating the electronic structure of a molecule and executed the code, it would generate a result at a given time. The scientist could then optimize the program to run faster or she could wait two years, buy the latest CPU, and assume that the program would generate the result twice as fast. Moore's law[1] gives the reason for this assumption, stating that the number of transistors on integrated circuits will double approximately every two years. This meant that the producers of CPUs could increase the density of the transistors which led to a higher clock frequency thereby making the chips approximately twice as fast. In 2006 the power wall[2] was hit as the increased power usage resulted in a higher temperature on the surface of the CPU making them more unstable. As a result, the chip producers needed a new strategy and instead of using the continuation in transistor to increase the frequency, the engineers decided to initial lower the frequency but add another core to the CPU and thereby the homogeneous multi-core CPU was born. Other CPU designers proposed other designs, some of which were heterogeneous like the CELL Broadband Engine (CELL-BE)[3] from Sony, Toshiba and IBM.

With the power wall problem solved thus solve, the result became that the users of scientific applications now had CPUs with lower frequency leading to longer execution time of existing applications or even worse a heterogeneous architecture, which could not execute their existing applications. Therefore, the scientists had to rewrite their applications in order to execute them in parallel on one of the new multi-core architectures. However, parallelizing existing applications is a difficult task even for experienced computer scientist and therefore too difficult for many non-computer scientists.[2] As a result, the need to design tools to assist these scientists in writing parallel code arose.

Throughout the years computer scientists, engineers and others has proposed many solutions to help scientists to write parallel code. These solutions range from automated parallelization to parallel programming languages.[4, 5, 6] However, the primary focusses of this thesis are a Distributed Shared Memory (DSM) model and a Communicating Sequential Processes (CSP) inspired model both for the CELL architecture as well as a data parallel library based on a DSM model for high level programming in Microsoft .NET names DistVES.

Finally, the thesis describes another Microsoft .NET solution namely a parallel model inspired by CSP for programming a mixed cluster environment named CloudVES.

Contributions

The first project of my PhD study was a system named "CSP to the CELL Broadband Engine". I did this work in collaboration with my colleague Kenneth Skovhede. The project was an extension of our master thesis in which we developed a DSM system for the CELL named DSMCBE. In spite of IBM discontinuing the development and production of the CELL architecture, we continued research on the CELL throughout the first year of the PhD program. Thereafter, I started on the DistVES system and about a year later, I concluded the work. During the last couple of months of the PhD program, I started working on a tool which allow transparent distribution of computational heavy methods to a cluster of workstations, servers, and cloud instances. Altogether, the three projects provide the central contributions to this thesis.

CELL The research involving the new heterogeneous multi-core CELL architecture from Sony, Toshiba and IBM started in late 2007, when Kenneth Skovhede and I together started work on our joined master thesis.[7] After a year, the research resulted in a DSM model supporting a cluster with multiple CELL units. The main idea of the project was to provide the programmer with a simple API and allowing her to think about the cluster as if it had one single large shared memory address space. This idea should simplify the writing of parallel programs that could fully utilize a CELL cluster.

The obtained results of the research inspired us to continue the work on the CELL architecture after we started in the PhD program in the spring of 2010. The first project was to develop a system for supporting the CSP paradigm on the CELL architecture and it involved two steps. Firstly, the design and implementation of a CSP channel model build on the DSM model from our master thesis. Secondly, implementing a model that would allow support for running multiple small user-threads on the specialized compute units of the CELL; thus, supporting CSP processes. Together the processes should communicate with one another through a number of channels.

The projects resulted in three papers (see appendix B1-B3) of which the first relates to the work done on DSMCBE during our master thesis.

DistVES The second project emerged from another joined project with Kenneth Skovhede in which we tried to support executing sequential .NET code in parallel on the CELL architecture and GPUs. However, due to difficulty identifying fine-grained parallelism in the sequential code we decided to stop the initial project. In spite of the setback, the project resulted in the idea of making a distributed version of one of the constructs namely the Parallel-For loop defined in the Microsoft Task Parallel Library (TPL). In late summer of 2011, I started on designing a system that would allow this construct to run on multiple machines instead of only running on a single shared memory machine as the Microsoft version does. This required that any shared object, used by a program, would be addressable across multiple machines yielding yet again a DSM approach. The project, named DistVES, resulted in one paper (see appendix B4).

CloudVES Late in the PhD program, an idea of using a combination of internal resources like non-utilized workstations, servers and external cloud instances to execute the computational heavy parts of a program arose. The distribution should be transparent for the users and should include a management tool along with security mechanisms, which would limit the transferring of sensitive data to external resources. Furthermore, as prices on external compute instances would become lower than an estimated price on running the internal servers, the scheduler should primarily schedule tasks to the external cloud instead of the internal resources, which could partly be shutdown. Due to time constrains, implementation was limited to only the most basic features; however, the project resulted in the submission of one paper (see appendix B5).

Publications

Morten N. Larsen, Kenneth Skovhede, and Brian Vinter. Distributed Shared Memory for the CELL Broadband Engine (DSMCBE). In Leonel Sousa and Yves Robert, editors, ISPDC, pages 121-124. IEEE Computer Society, 2009.

Kenneth Skovhede, **Morten N. Larsen**, and Brian Vinter. Extending Distributed Shared Memory for the CELL Broadband Engine to a Channel Model. In Kristján Jónasson, editor, PARA (1), volume 7133 of Lecture Notes in Computer Science, pages 108-118. Springer, 2010.

Kenneth Skovhede, **Morten N. Larsen**, and Brian Vinter. Programming the CELL-BE using CSP. In Peter H. Welch, Adam T. Sampson, Jan Bækgaard Pedersen, Jon M. Kerridge, Jan F. Broenink, and Frederick R. M. Barnes, editors, CPA, volume 68 of Concurrent Systems Engineering Series, pages 55-70. IOS Press, 2011.

Morten N. Larsen, and Brian Vinter. Distributed Virtual Machine for Microsoft .NET. In Journal of Software Engineering and Applications 2012, 5(12):1023-1030.

Morten N. Larsen, and Brian Vinter. Transparent offloading of parallel components in Microsoft .NET. *Submitted to "Journal of Cloud Computing: Advances, Systems and Applications" on the 8th of March 2013.*

Outline

As mentioned above, systems exists to help programmers write parallel code. Chapter 1 introduces these systems along with the concepts of Parallel Programming. Furthermore, the chapter introduces Microsoft .NET, the Microsoft Parallel Library and describes the different classes of high performance applications based on their characteristics. Chapter 2 presents the work done on the CELL architecture together with Kenneth Skovhede. Chapters 3 and 4 gives a detailed description of the research on the Distributed Virtual Execution System and the transparent distribution of tasks to a mixed platform. Chapter 5 concludes on the findings along with some perspectivation. The publications resulting from this PhD study are placed in their full length in appendix B and finally appendix A contains all references.

Chapter 1

Parallel Programming

As described in the introduction the concept of parallel programming became a necessity as the multi-core CPU architecture gained a wide footing in personal computing. As, it was stated in the introduction parallel programming is difficult, and to underline that statement, the following section will describe some of the most critical design issues that the programmer must consider before programming parallel code. By the end of this chapter, it should be clear why parallel programming is so challenging and why it is necessary to develop tools for assisting the programmer in writing parallel code.

1.1 Designing Parallel Programs

If a person have a sequential version of a program and wants it to run in parallel, a common assumption is that most of the design decisions for the parallel version is the same as for the sequential version. Thus, a rewrite of the central components using threads to parallelize on a single machine is necessary. In some cases, normally only in the very simple ones, this would be a legitimate assumption, but in most cases the parallel version will not scale linearly. Therefore, the programmer must often rewrite the essential components completely and rebuild the program by making new design choices regarding target architecture, data distribution, communication, level of parallelism etc. The rest of this section assumes that the programmer wants to write a parallel program without using any of the available tools meant for helping the programmer in doing so.

1.1.1 Target architecture

The basic architecture for a system capable of executing parallel code is typically an interconnected collection of computers having one or more processing units. In addition, a target system can also consist of a single machine with multiple cores. The processing units of the machine(s) in the system can either be a homogenous architecture like in a normal workstation with e.g. a x86 processor; a heterogeneous architecture like the CELL-BE; or even two homogenous architectures combined like in a normal workstation equipped with a graphics processing unit supporting general-purpose computing (GPGPU) making it a heterogeneous system. The homogenous machine is the simplest to program, because the architecture consists of one or more similar processing units and normally a single memory system. The heterogeneous machine consists of multiple types of processing units and/or multiple types of memory systems each with different characteristics, instruction sets etc. As these systems have different properties, the programmer must decide on a target architecture from the start, as a unified design that works perfectly on both homogenous and heterogeneous architectures is difficult to define and more importantly very rare unless one uses a tool like Message Parsing Interface (MPI) for handling the underlying hardware.

1.1.2 Memory

When having decided on a target architecture, the next step is to look at the available memory models supported by the given architecture. This section describes the three most common memory architectures where implementation in both hardware and software is possible. The combination of different memory models is also possible. One example being a software implementation that handles communication between multiple hardware implementations of a shared memory model resulting in distributed shared memory.

Shared Memory

Most modern multi-core architectures (symmetric multiprocessing)[8, 9] have random access memory (RAM), typically shared between all the computational cores. Thereby all cores in the system have the same view of the memory (see figure 1.1). This concept of sharing memory between all cores is named shared memory[10] and the term is furthermore used when talking about software threads sharing a memory region with each other. The advantages of shared memory is that it gives fast intercommunication between multiple threads running on perhaps different computational cores. In addition, it is very simple to program as all cores/threads uses the same memory address space.

However, shared memory systems are not usable for massive parallelization, as it does not scale beyond a dozen of computational cores. The reason for this is the memory wall,[11] meaning that the clock frequency of the CPU is higher than the clock frequency of the memory and as a consequence getting data from the memory to the computational cores becomes the bottleneck. One way to minimize this problem is to make small caches close to the computational cores. The performance of these small caches are higher than the RAM and therefore they can be used to store frequently accessed data close to a given core. Furthermore, by dividing caches into levels where each core has a level one cache that can only be access by the given computational core and where a level two cache would be shared between two cores and the level three between four cores an so on, allows computational cores to share caches. In spite of this design, caches also add problems when a core updates a value in the cache as

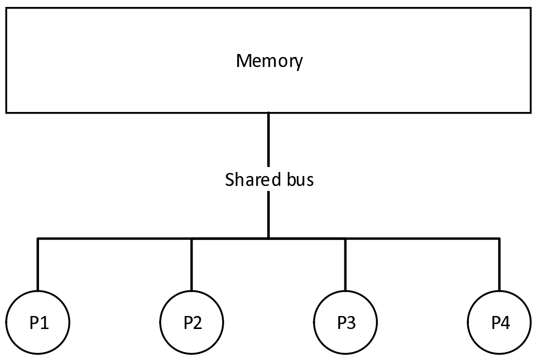


Figure 1.1: Overview of shared memory architecture with four processors (P1-P4)

the cache must communicate the update to all other caches that holds a copy of the value in order to guarantee cache coherence. Therefore, shared memory in the current design is not going to scale to homogenous architectures with more than ten to twenty cores.

When designing an algorithm it is sometimes necessary to consider the time required for a given computational unit to access a given memory location. This is especially the case when some computational units can access a part of the memory faster than others. One example is if the first computational unit can access the first X bytes of memory ten times faster than the rest of the memory and the second computational core could access the following X bytes ten times faster. Compared to if all computational cores could access all the memory in the same amount of time, the two systems would yield two different approaches when making a design of a parallel program. The term "uniform memory access" (UMA), is used to define that the time it takes to access any single memory location is the same for all cores; however, the private caches of the computational units may make some memory accesses faster than others, if the cache holds a valid copy of the given memory location. Therefore, the effect of caches most also be taken into account when designing an algorithm.

Distributed Memory

Due to the scalability problems with shared memory, another solution namely distributed memory was proposed. Distributed memory is essential a number of memory/CPU pairs e.g. interconnected shared memory machines (see figure 1.2). By this alternation, programmers can no longer think of the memory as one large address space but should see the system as several address spaces. This naturally makes it more difficult for the programmer, but that is the price for scalability to hundreds of machines. However, a skilled programmer will be able to design their algorithms in such a way that the distribution of data over the available memories is simple and at the same time very efficient. Consequently, this model forces programmers to consider which data distribution would result in the best performance. In distributed memory, the computational cores access-time to any memory location is not uniform and the term non-uniform memory access (NUMA) describes this.

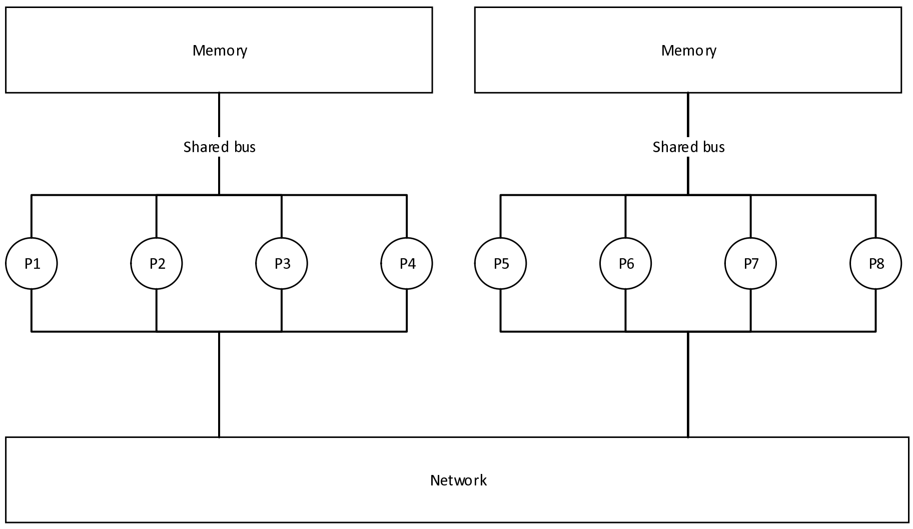


Figure 1.2: Overview of a distributed memory architecture with two machines.

Using explicit message parsing makes it possible for the different memory regions to communicate. Having said that, there exists system like the MPI[12] model for hiding the writing of communication code from the programmer. Unfortunately, MPI does not relieve programmers of considering the distribution of data. Because, whether the programmer chooses an explicit or implicit message parsing model, a poor data distribution often yields poor scalability.

Distributed Shared Memory

Having a system, which is easy to program and another system, which scales well, gives way for a merge of the two system into one i.e. Distributed Shared Memory (DSM). Like the distributed memory model, DSM has NUMA properties and the systems normally implements the model by defining a global address space that maps all the involved memories into one. In addition the model use implicit message parsing to allow accessing memory location on other machines. However, it is still necessary for each participating machine to map a given global memory address to a specific memory location on a given machine. Over the years scientists has proposed solutions ranging from a central server solution with total overview of the mapping to totally distributed solutions like the directory model used in some hardware implementation of DSM like DASH.[13] The DSM model has over the years also been implemented in software like Munin,[14, 15] ORCA,[16] Midway.[17, 18]

Distributed shared memory has the advantage that it hides the message parsing from the programmer; unfortunately, it does not hide the latency. Furthermore, the hiding of the underlying architecture does not urge the programmer to think about how the algorithm should distribute data in order to get optimal performance. Therefore, the question is which model is best for the average scientist, and the simplified answer is that it is better to get some parallelization using a DSM model than using a too advanced model,

Table 1.1: Bandwidth and latency for memory and different network types.

	Bandwidth (Gbit/s)	Latency
Memory	100-192	10-100 ns
1 Gbit	1	25 μs
10 Gbit	10	2-4 μs
Infiniband 1x	16	1-3 μs
Infiniband 4x	300	1-3 μs

which the average non-computer scientist does not understand or is incapable to implement in the algorithm.

1.1.3 Communication

Communication in a parallel system exists on two levels. The intra-communication takes place inside a single machine between the computational cores and the memory and inter-communication between machines. Intra-communication is in most cases a bus with high bandwidth and low latency, whereas the inter-communication is over a network with low-bandwidth and high latency compared to the bus. Table 1.1 shows bandwidths and latencies for some of the most common memory/network types. The values shown are the maximum theoretical values and especially bandwidth is often somewhat slower in practice.

As seen in table 1.1, memory is supreme compared to network both in terms of bandwidth and in terms of latency. Consequently, it is important that the programmer considers the distribution of data, so that data is close to the computational unit wanting to use the data. Furthermore, if the bookkeeping needed to manage the data distribution is low, the network usages will decrease. However, some algorithms are very depended on a lot of communication. Therefore, it can sometimes be more efficient to transfer the code to the machine having the data in memory instead of the other way around. These considerations does not directly apply for shared memory, as all the data is on the same machine and for the distributed memory with explicit message parsing it is more evident to consider data distribution. Regarding, distributed shared memory it is easy to skip the data distribution considerations, as shared memory and distributed shared memory for the programmer has many of the same properties and the implicit message parsing handles the actual data transfers.

1.1.4 Levels of parallelism

This section will describe the different levels of parallelism seen in common algorithms.

Instruction-level parallelism Instruction-level parallelism is the lowest level of parallelism. A simple example is the following calculation $A = (B + C) \cdot (D - E)$ which consists of the following three operations:

```
    Temp1 = Add(B,C)
```

```
Temp2 = Sub(D,E)
A = Mul(Temp1, Temp2)
```

It is possible to execute the Add and Sub operations in parallel as these two operations each consist of two reads followed by a write and none of them accesses the same variables. On the contrary, before executing the final multiplication the previous two operations must finish. One can use instruction parallelism for pipelining and in some cases it is possible to convert for-loop parallelism (see data-level parallelism below) into instruction-level parallelism by unrolling the loop. Normally the level of parallelism obtained by using this type of parallelism is relatively low.

Data-level parallelism Data-level parallelism, also named loop parallelism, defines the execution of the same operation(s) on all the data elements at the same time. Below a simple example illustrates data-level parallelism:

```
int[] A, B, C
for(int i = 0; i < A.length; i++)
    A[i] = B[i] + C[i]
```

Here the execution of the single addition inside the loop on all elements of the arrays at the same time is possible as long as there is no dependencies between the arrays. However, if we changed the operation inside the loop to `B[i+1] = B[i] + C[i]` and change the loop accordingly to avoid loop overrun, then the code could no longer be parallelized to the same degree, as the i calculation is now in conflict with the $i + 1$ calculation. Some algorithms are naturally data-level based, one example being parameter sweeping. One can use parameter sweeping for finding the optimal input to a given function based on some user-defined criteria. An easy way to achieve this is to try all possible continuations of the input parameters to the function. Each combination of input parameters produces a task, and the system can afterward schedule the tasks among the available computational units in a given system.

Task-level parallelism A third way to do parallelization is the use of tasks. A simple definition of tasks is that a task is a list of instructions along with input data that produces an optional output, e.g. the result of executing the instructions. In contrast to data-level parallelism, the use of task-level parallelism is to do multiple (normally different) tasks on the same data set. A simple example is to find the maximum and minimum values on the same data set and it is clearly possible to execute these two tasks in parallel.

1.1.5 Maintaining correctness of an Algorithm

Writing parallel code is a demanding task that requires a good overview of the executing architecture as well as in-depth understanding of data distribution. Therefore, the programmer must understand and be aware of the basic pitfalls that will arise when doing parallel programming and the rest of this section describes some of these pitfalls.

Protecting critical regions Writing a parallel program is in most cases more difficult than writing a sequential program. The reason for this is that the programmer must consider that multiple units can access shared data concurrently. In a program where all data access is write-only or read-only, the programmer does not need to worry about concurrent data access, because two units reading or writing the same variable is not a problem. However, such programs are rare and normally not very useful. A simple example of a read-/write data access is the non-atomic increment instruction illustrated in figure 1.3, which reads a variable, then adds one to the value and writes the result to the same variable. This operation is not easy to do in parallel because both units will read the same value, add one, and then write the same value to the variable thereby increasing the value with one instead of two. Many other examples exist, and the programmer must identify these problems when writing the program in order to get a program that will produce the correct result. The term critical region defines areas of code with these kind of instruction patterns and programmers must protect these regions using locks or other kinds access mechanisms i.e. mutexes, semaphores, barriers, monitors. All of these mechanisms allow only one unit inside the critical region at the same time, meaning that only one unit at a time can access a given shared variable. Critical regions are also found in data structures like hash tables, lists, queues and all accesses that modifies the content of these data structures requires some sort of locking mechanisms. There exist data structures, which does not require protection or have one build-in and we define these as thread-safe data structures.

The problem with protecting critical regions is that it results in sequential execution, so the programmer must try to minimize the use of the locking mechanisms or rewrite the program to use data structures or data patterns which does not require locking. In addition, the overhead of using locks is significant.

Deadlocks As describe previously there is a need for locking mechanisms to ensure correct program execution, but unfortunately, locks also gives the programmer a new concern. In a program with multiple locks, there is a risk of dead lock, meaning that two execution units wait on the lock that the other unit currently holds. Figure 1.4 shows how a unit first takes lock #1 and at the same time, another unit takes lock #2. Afterwards the second unit tries to take lock #1, but has to wait until the first unit releases the lock. In the meantime, the first unit try to take lock #2, but also has to wait until the lock is release by the second unit. Now none of the two units can continue execution, and there is a dead lock in the program execution.

Several methods are available for detecting dead locks, and methods also exist to solve deadlocks at runtime.[19] Most of these methods only work at simple problems as detecting and/or solving deadlocks are a NP-hard problem.[20]

Race conditions Another risk when writing parallel programs are race conditions. This kind of data race is very hard to spot and can easily exist in programs that has run many times; however, suddenly the program fails or produces a wrong result. The reason for these races are typically that the programmer has failed to identify one or more critical regions, resulting in two processes accessing shared data simultaneous. In most cases, the program

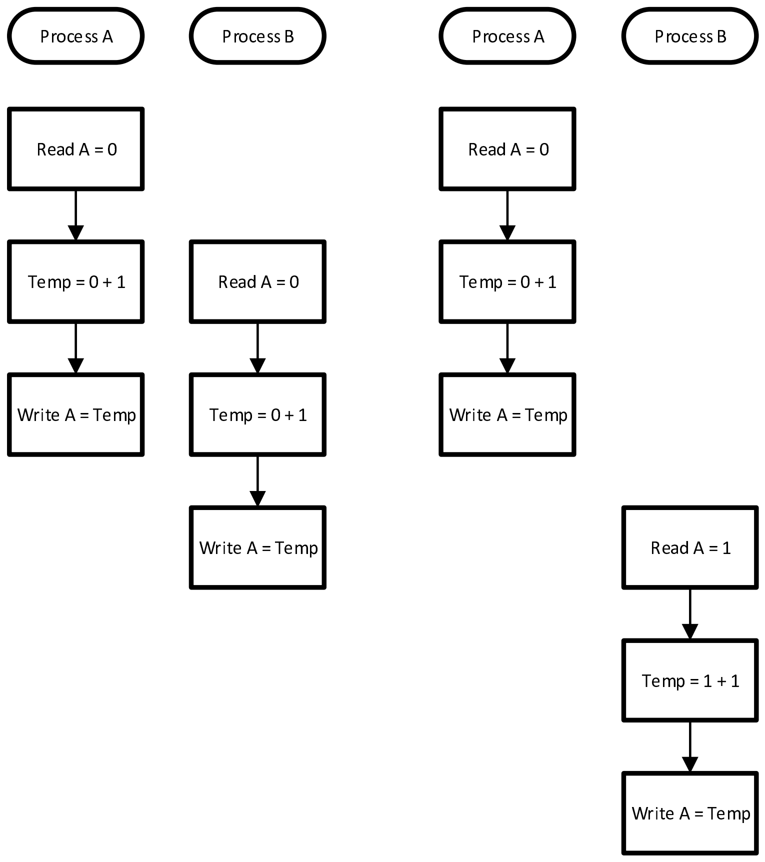


Figure 1.3: The left side of the figure shows run where the critical region is not protected, whereas the right side is.

runs as expected, but suddenly a small shift in the execution order due to e.g. thread scheduling or changes in the input data. This changes the program execution order and now a race condition arises and the program fails in some way without the programmer having any idea why, because the program ran correctly in the previous run.

1.2 Systems to help to programmer writing parallel code

This section will describe some of the systems for helping the programmer write parallel programs that I have found during the PhD project. The presentation will start with a short introduction to the Microsoft .NET platform followed by a description of the Microsoft Task Parallel Library. This should form a basic understanding of the platform used to design and implement the DistVES and CloudVES systems described later. Afterwards the section continues with other widely used or interesting systems that target different architectures, different types of parallelism and/or different classes of programs.

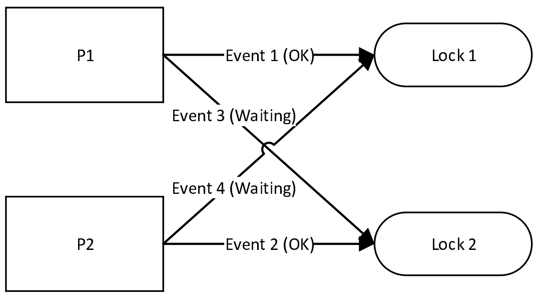


Figure 1.4: Two processes both needing to take both Lock 1 and Lock 2. P1 first takes and hold Lock 1 and at the same time P2 takes and hold Lock 2. Afterwards, P1 tries to take Lock 2 and P2 tries to take Lock 1.

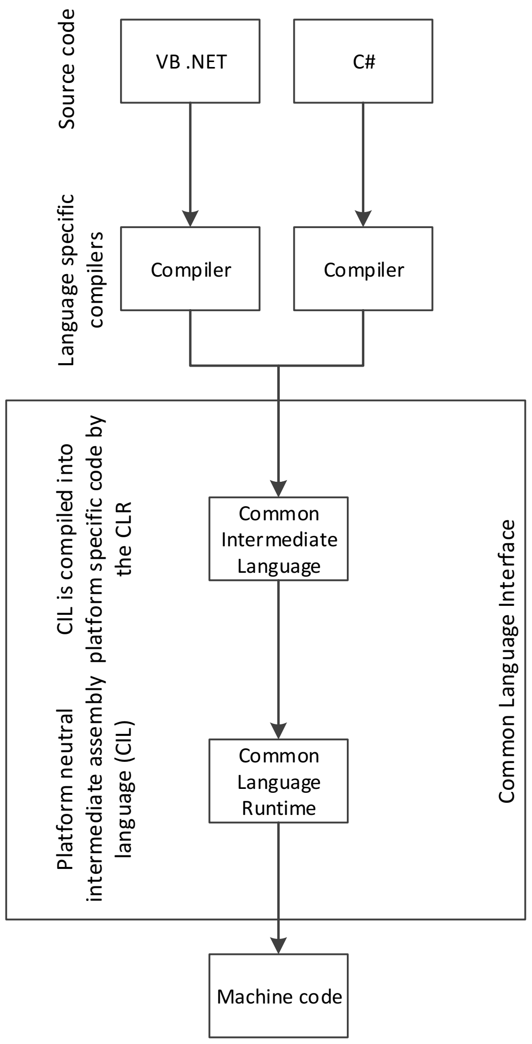
1.2.1 Microsoft .NET and Parallel Extensions for .NET

Microsoft .NET is one of the software frameworks available for Microsoft Windows and the framework supports a number of programming languages among others C#, F#, Virtual Basic, Virtual C++. Furthermore, the framework contains a large library containing functionality for a long list of commonly used methods for networking, cryptography, threading, data containers etc. The compiler compiles the source code written in one of the supported programming languages into the Common Intermediate Language (CIL), which is an object-oriented, stack-based assembly language. Using a shared assembly language allows calling methods written in other of the supported programming languages meaning that a method written in C# can be called from code written in Virtual Basic. On a given architecture, the Common Runtime Language (CLR) Just-In-Time (JIT) compiler compiles the CIL to the machine language supported by the architecture. This happens inside the Virtual Execution System (VES) that is the application virtual machine of Microsoft .NET handling execution of .NET programs. The the main responsibilities of the VES is to handle security, memory management including garbage collecting, exception handling. Figure 1.5 giver an overview of the .NET pipeline.

An open-source project of Microsoft .NET named Mono exists and allows execution of programs written in one of the programming languages to run under Linux. Mono supports most of the functionality and libraries available in .NET; however, the implementation of the methods may differ and therefore there is a risk of different results depending on whether Mono or .NET executes the code.

Integrated into the fourth version of Microsoft .NET was a new component called Parallel Extension, which consists of the Task Parallel Library and PLINQ (Parallel LINQ). This section will give a short description[1] of the first of these two components, whereas the latter is not related to the rest of the work of this thesis and therefore not described. In order to make it easier for programmers to execute code in parallel, Microsoft introduced the task Parallel Library. In addition to supporting task parallelism, the task library also supports data parallelism. The library supports Task-Parallelism using

[1]See [21] for a more detailed description on how the library could be implemented.



Source code
VB .NET
C#
Language specific compilers
Compiler
Compiler
CIL is compiled into platform specific code by the CLR
Common Intermediate Language
Platform neutral intermediate assembly language (CIL)
Common Language Runtime
Common Language Interface
Machine code

Figure 1.5: An overview of the .NET pipeline

the Parallel.Invoke construct and data parallelism through the two parallel constructs Parallel.For and Parallle.ForEach.

Microsoft implements the data parallel methods using replicable tasks, meaning that the scheduler replicates each task to all available workers and then the workers executes each task several times until the For-loops condition variable is satisfied. This naturally requires that all cores can read and increment the counter variable of a given loop. Microsoft suggests that this could be implemented using the atomic compare and increment method in the Interlocked library. The method ensures that only one worker at a time can access the counter. Unfortunately, the use of this CIL operation means that the Microsoft Parallel system only supports a single machine. The advantages of this design is that it is simple and highly de-centralized as the work thread does not need to communicate with a centralized unit and/or each other. Therefore, the parallel loop should scale in a very efficient manner.

The library uses the compiler to rearrange the code in order for multiple threads to access the shared variables. In addition, the library contains a scheduler that is responsible for assigning tasks to the worker threads running on each of the available computational cores. We base the design on each worker thread having a queue with tasks that it should run. If a worker thread, has executed all the tasks in its queue, it can steel tasks from the queues on the other work threads and the term “work stealing” defines this behaviour and is known from the Java fork-join framework[22] and Cilk+.[23]

1.2.2 Automatic Parallelization

During the last decades, the number of developed tools for full automatic parallelization of sequential programs has been high, but the success rate of these tools has been low. Today it is widely accepted that, , with the exception of very simple sequential programs, full automatic parallelization of sequential programs is if not an infeasible solution then at least many years away. The main problem is that it is difficult to identify and extract the parallelism within a “random” sequential program. Furthermore, the compilers need knowledge about input parameters etc. in order to reason about the parallelism of the sequential program. In the end, it is very difficult for the compiler to make new data structures and distribute data in way that would result in good scalability of the sequential program. For the programmer auto parallelization would, obviously be the optimal solution, as it would require no work except recompiling to get a program that could run in parallel. The target for auto-parallelized programs are often single machines with multiple cores, but system exists that targets multiple machines.

1.2.3 Remote Procedure Call

Instead of having a distributed shared memory system, one could use remote procedure calls (RPC).[24] The idea is that the programmer can transparently execute a method on another machine instead of running it locally. The RPC system will manage the packing, transferring and unpacking of parameters and method information needed to execute the method on the other machine. Most programming languages supports RPC, however, the practical part of connecting clients and setting up the RPC back-end, can sometimes be cumbersome, but writing programs using RPC is simple. The performance and scaling of RPC is acceptable for smaller systems, but for large system it will in most cases not be enough especially is the program makes many small calls.

In .NET there is also a system for doing RPC on distributed systems named .NET Remoting.[25] However, today, most programmers use Windows Communication Foundation (WFC)[26] for doing RPC. In addition, Windows defines “Proxy objects” and such an object is a client-side version of the correspondingly object on the server-side. Modifications made on the client side are transparently executed on the server-side.

1.2.4 Message Parsing Interface (MPI)

The Message Parsing Interface is a model intended for assisting a programmer in handling communication and data transfers in a parallel system preferable consisting of a homogenous systems with multiple machines connected through an interconnect.[12] Currently, MPI is one of the most used models for writing programs made for large parallel systems and a simple API and high portability are the base for its success. Distributed memory architectures is the main target for MPI and is preferable used for programs with the Single-Process-Multiple-Data (SPMD) property; meaning that the computational cores executes the same process for each of the multiple data items. The model includes management of nodes in the system, sending messages between nodes, allowing the programmer to define MPI ready data types, aggregating functionality like reduce, broadcast, and direct read/write of remote memory. Today most of the commonly used programming languages such as C, C++, Java, Python, Fortran have libraries that implement support for MPI.

One of the problems with models like MPI, and most other tools for helping the user write parallel code, is that it requires the programmer to learn a new model and then re-implement the algorithm that the user wishes to run in parallel. However, if the programmer is capable of learning the model and re-writing the algorithm, then there is a high possibility for getting good scalability and high performance. Nevertheless, like discussed earlier, MPI hides the underlying hardware architecture completely from the user and therefore there is a risk of the programmer rewriting the original program without considering the best possible data distribution given the available hardware.

1.2.5 Intel Task Building Blocks

The Intel Task Building Blocks (TBB) is a C++ template library for helping the programmer write parallel programs without the programmer having to worry about threads.[27] Instead, the library provides a list of parallel constructs like Parallel_For, Parallel_Reduce, Parallel_Pipeline and data structures that supports concurrency. Furthermore, Intel TBB supports memory allocation in a scalable manner along with different kinds of mutexes and atomic operations. Altogether, this means that the programmer can focus on implementing the algorithm instead of handling threading and furthermore she does not have to think about the underlying operating system and machine architecture, as the Intel TBB system will handle these things.

Behind the scenes, the library has a scheduler that ensures distribution of tasks in an “unfair” way, because Intel TBB prefers efficiency to fairness. In addition, the system tries to ensure cache affinity, meaning that scheduling will try to ensure that when a given task is running on a processor the data in the cache is the data used by that task. In addition, if the same task executes multiple times it is preferred that it runs on the same processor. The scheduler is using the work-stealing concept also known from Cilk+ and Microsoft Task Parallel Library to distribute tasks to the available computational units on the machine. However, Intel TBB does not yet support multiple machines.

1.2.6 Intel Cilk Plus

In the Cilk+ system, which was later adopted by the Intel Corporation, , the programmer is responsible for identifying the code that the system can safely execute in parallel. This is in contrast to the auto-parallelization discussed above.[28] The system should then be responsible for handling distribution of data and scheduling of work to the available cores in the system. The code below shows the syntax for using Cilk+:

```
cilk int fib(int n)
{
    if (n < 2)
        return n;
    else
    {
        int x, y;

        x = spawn fib(n-1);
        y = spawn fib(n-2);

        sync;

        return (x+y);
    }
}
```

The `spawn` method will spawn a new procedure and the keyword `sync` is used for making a synchronization point for the created procedures. Furthermore, Cilk+ support inlets with implicit atomicity around them, which simplifies reasoning about concurrency and non-determinism. Along with inlets, there exists an abort method to abort an inlet procedure.

The scheduler is, as with many of the systems for parallel programming, the most interesting part and the design of the work-stealing scheduler in Cilk has been an inspiration to the schedules used in many others systems like the Microsoft Parallel Library and Intel TBB. In contrast to a traditionally centralized scheduler, the work-stealing scheduler is Cilk+ base so each computational unit is responsible for handling scheduling locally thereby making it more distributed. However, a centralized part will distribute the tasks in a round-robin fashion to all available workers. This would normally result in a very unbalanced execution, as some task would require little work and some much more. It would eventually mean that some computational cores would complete all the tasks in their work queue quickly while others would work for a longer time, as they randomly had received some task, which required more work. Cilk+ solves this problem by allowing workers to steel work from one another. So when a given worker no longer was any work it will simply choose a random worker and steal a task from its work queue. This will lead to a much more balanced execution; however, the risk of some workers doing more work than others remains. To avoid heavy locking to protect the work queues, a double-ended queue makes the basis for the implementation, where the worker thread de-queues from the top and threads stealing from a queue take elements from the bottom. This design ensures that the worker thread does not have to lock the queue every time it access the queue; however, the stealing threads always have to lock to avoid other stealing threads to access the queue simultaneous. Only in situations in which there are very few elements in the double-ended queue must the worker thread lock the queue.

Finally, Cilk+ creates two versions of the same code namely a fast sequential version without bookkeeping and a "slower" version with bookkeeping enabling data to be transferred between units. The idea is that when the same computational core runs a given task it can use the fast version, but when another computational core steals work, the bookkeeping ensures data concurrency and allow data transfers as the task possible depends on the results of the child tasks.

1.2.7 Open Multi-Processing (OpenMP)

Another method for writing parallel code is OpenMP were the programmer use annotations to make existing code run in parallel.[29, 30] The focus of OpenMP is mainly on parallelization of loops. The compilers to the C, C++ and Fortran are some of the programming languages supporting the OpenMP annotations. OpenMP use a Fork-Join model with a Master thread which can spawn (fork) worker threads, which will eventually join and then the master thread will continue execution (see figure 1.6). Data transfers between the master thread and the worker threads are transparent to the user, so are synchronizations; however, the system supports that the programmer defines synchronization points using barriers etc. OpenMP supports systems ranging from simple desktop solution to large systems with hundreds of computers, but the scalability of loop parallelism limits the overall scalability. Below is a short example of using OpenMP.

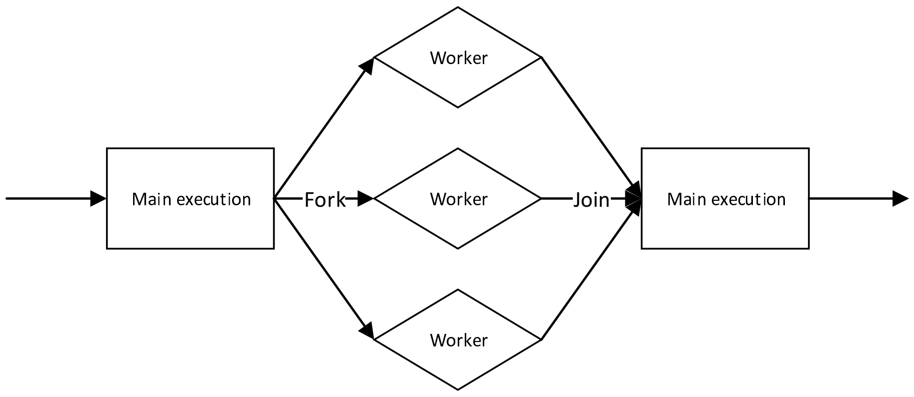


Figure 1.6: The Fork-Join principle

```
int main(int argc, char *argv[]) {
    const int N = 100000;
    int i, a[N];

    #pragma omp parallel for
    for (i = 0; i < N; i++)
        a[i] = 2 * i;

    return 0;
}
```

1.2.8 MapReduce

In 2004, Google presented the MapReduce systems developed to handle the forever-increasing amount of data in their datacentres.[31] The name MapReduce comes from the Map and Reduce methods used in many functional programming languages. The idea is that given a huge data set the Map function groups data into buckets in which all the data in a given bucket has the same properties as specified by the Map function. Afterwards the Reduce function execute a method on each of the buckets. A simple example is to use e.g. shipping documents from Amazon to find the number of times Amazon sold any given item. Given that all documents contains a list of pairs (item, quantity), the Map function map all pairs with the same key into the same bucket. For each bucket the reduce function has to accumulate the value of each of the pairs producing the result list containing pairs (item, total quantity).

Behind the scenes, the MapReduce framework first have to read data from solid storage i.e. from the hard disk drive or tapes and then produce the initial pairs (item, quantity). Afterward, MapReduce uses the Map function to group the initial pairs and the system afterwards distribute the result to all the reducers using a partition function. The system then sorts the results using a comparison and executes the reduce functions on each pair. The final step is to store the result in a distributed way.

Today, many use MapReduce as the amount of data on the internet is ever growing and the desire for doing data mining is increasing. The MapReduce has since 2004 been implemented in other frameworks supporting other programming language like Hadoop for Java.[32]

1.3 High Performance Applications

Most applications can be divided into different categories depending on their characteristics and most parallel applications fall into one of the following five categories, i.e. compute intensive, data intensive, communication intensive, memory intensive, or continuous service. Below is a description of the five categories along with some of the application types that typically fall into the given category. Systems for assisting the programmer in writing parallel code will typically focus on applications that fall into only one or two of the five categories. The reason for this is that developing a single system that will solve all applications in any of the categories is incredible hard because of the categories very different characteristics.

1.3.1 Compute intensive

The first category is the compute intensive applications and as the name implies the focus is doing calculations. In order to do the calculations the application “uses” numerous CPU cycles meanwhile the amount of data used from disc or memory is very limited. Monte Carlo simulations along with parameter optimizations are typical applications of this category. Furthermore, applications in this category are normally simple to parallelize and scientists use the term “embarrassing parallel” to define these applications.

1.3.2 Data intensive

Another category is the data intensive problems. This type of applications uses a large amount of data from solid storage like disc or tape. Typically the applications are signal/image processing, pattern matching, data mining. The problem with the applications in this category is that they need to get the data from I/O very fast in order to scale. The solution could be SAN servers for storage and a MapReduce model for data mining or machine learning.

1.3.3 Communication intensive

The third category is the communication intensive problems and the algorithms in this category have an intensive need to communicate with each other. The category contains algorithms for particle physics (successive over-relaxation problems) and many of the algorithms implemented with MPI have the high communication characteristic. For improving the execution time for this category, the systems requires a fast interconnect like 10 GB Ethernet or infiniband.

1.3.4 Memory intensive

The fourth category is the memory intensive algorithms and many algorithms fall into this one. One of them could be DNA assembly and the goal is to minimize the time required to access memory locations. However, normally the memory used by these algorithms are much larger than a single machine can handle. Therefore, programmers work a single but very large shared memory is beneficial. Thus, the applications requires a large computer with distributed shared memory implemented in either hardware or software.

1.3.5 Continuous service

Lastly, there is the category of continuous service applications and in this group belong amongst others databases and web-servers. The requirement for these applications is that they must be accessible 24 hours-a-day. In addition, the system should dynamically adjust the amount of executing hardware so it will match the current workload of the given application. Therefore, dynamic scalability and fault tolerance is essential in the design and such systems could be cloud based or based on virtualization software like VMware.

1.4 Summary

This chapter describes some of the challenges of writing parallel programs and/or back-end systems for doing so. In combination with the multiple research on this topic,[33, 34, 35] it is clear that writing parallel programs is a difficult tasks and requires a thorough understanding of the underlying hardware, the available programming tools, and models, in order to achieve a good performance. Furthermore, the process of debugging parallel programs is very cumbersome, as most IDEs does only have limited support of doing so, especially if one wished to debug the code in multiple machines.

Chapter 2

The CELL Broadband Engine

In the year 2007/2008 the CELL Broadband Engine (CELL-BE)[3] was one of the hottest architectures on the marked. Therefore, it was an obvious target platform when I together with Kenneth Skovhede started looking for an interesting master's project in the field of distributed and parallel computing. Some years later when we started our PhD studies, IBM had already announced that they would discontinue the CELL-BE architecture, but we continued our collaboration on making it simpler for programmers to write parallel programs for the architecture using Communicating Sequential Processes (CSP).

This chapter will introduce the architecture and give a short description of our master's project. There after, follows a more in-depth description of making CSP support for the CELL-BE architecture. Skovhede and I divided the project into two parts; namely a channel part and a processor part. The following two sections describe the work followed by a conclusion. All the work described in this chapter was done in collaboration with Kenneth Skovhede.

2.1 Design of the Architecture

To solve the problems with the traditional Von Neumann architectures Sony, Toshiba and IBM (the STI alliance) introduced the CELL-BE architecture in the mid-2000. The new architecture targeted both the gaming industry i.e. the Sony PlayStation 3 as well as high performance computing industry. The rest of this section describes the most essential details of the architecture.

2.1.1 Overview

The architecture is two-folded meaning that it consists of two types of computational units with different responsibilities, functionality and instruction sets. The main computational unit is a traditional 3.2 GHz twin threaded PowerPC which is responsible for running the operating system, handling all I/O, coordinating operations running on the specialized computational units.

IBM originally designed the PowerPC[36] and over the years, it has been the central computational unit in many high performance computers as well as in some workstations. The PowerPC uses the Reduced Instruction Set Computer (RISC)[37] instruction set in contrast to the x86/x64 instruction set used in most of the processors from Intel and AMD.

The second computational unit of the CELL-BE is the Synergistic Processing Element (SPE). Each of the eighth SPEs available in a standard CELL-BE chip consists of a processor named Synergistic Processing Unit (SPU), a Memory Flow Controller (MFC), and a small 256 KB Local memory Storage (LS) for storing code and data. This computational unit can executed a simplified RISC instruction set and has a performance of 25.6 GFlops when doing single precision float operations. To get such high performance the SPUs are very simplified and does not contain caches, branch prediction etc. However, the SPU is capable of during 128 bit Single-Instruction-Multiple-Data (SIMD) instructions, making it possible to execute four instructions like additions (single precision) in parallel. Figure 2.1 gives an overview of the design of the SPEs. The SPE cannot access data directly from main memory, only data placed in the LS is accessible. However, the MFC can transfer data between main memory and the associated LS using Direct Memory Access (DMA) transfers. Each MFC has a queue with 16 placeholders of current DMA transfers and the transfers can run fully in parallel with the given SPU doing calculations. This design allows programmers to execute code on the available data while transferring the data needed by the next execution to the appropriated SPU. The result is that it is possible to hide most of the data transfers, allowing for full utilization of the SPUs. Finally, communication between the PowerPC and the SPEs occurs by sending messages explicitly and/or by using some of the interrupt calls supported by the system. It is also possible to send messages internally between the SPEs.

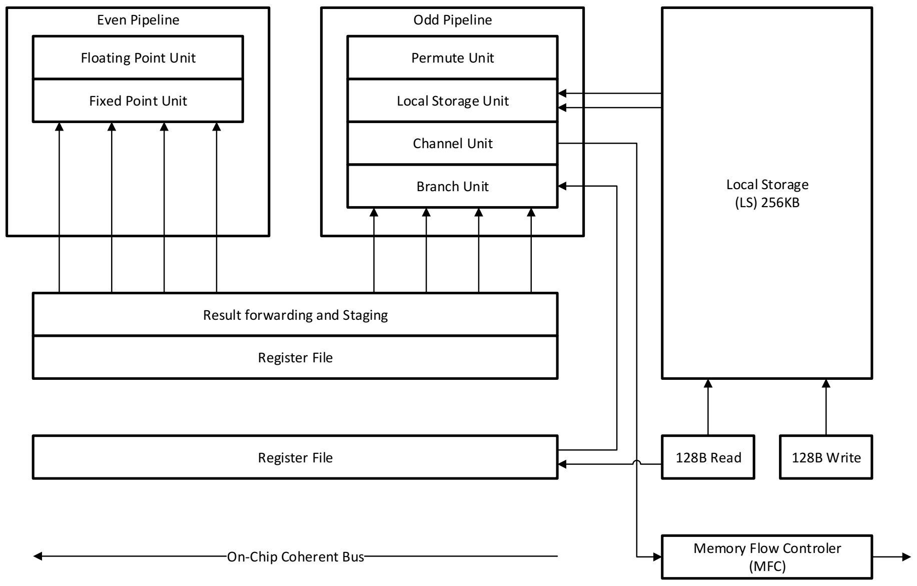


Figure 2.1: Overview of a Synergistic Processing Element (SPE).

A specialized bus named Element Interconnect Bus (EIB) connects the PowerPC processor and the SPEs with the main memory module. The EIB has a theoretically bandwidth of up to 204.8 GB/s at half the clock frequency of the PowerPC. The EIB is designed as a ring with the different components attached, allowing up to eight simultaneous data transfers. Figure 2.2 illustrates the design of the bus and as shown, each SPE can handle one incoming and one outgoing transfer at the same time.

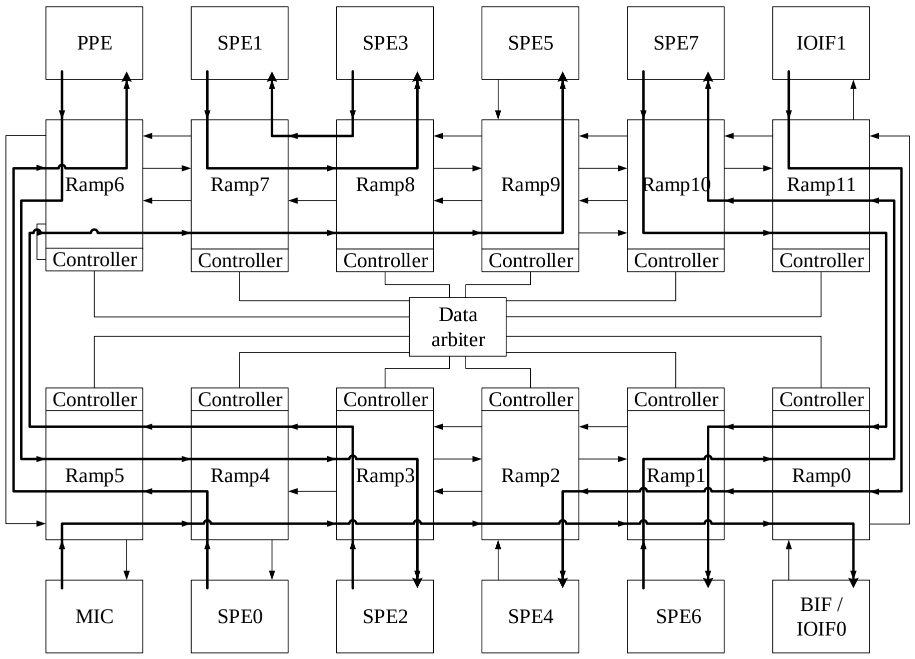


Figure 2.2: Overview of the Element Interconnect Bus (EIB) running sixteen simultaneous data transfers.

As mentioned earlier, the STI alliance produces the CELL-BE for both gaming and high performance computing. However, in order to guarantee low production cost, the manufacture ensures, by possible disabling one of the SPEs, that the PlayStation 3 only have seven SPEs. When used for high performance computing, it is possible to join two CELL-BE chips giving a total of sixteen SPEs and two PowerPCs. The IBM BladeCenter allows multiple of these elements to connect through Ethernet or infiniband making a powerful cluster. In order to lower the cost of making a cluster, some scientists has made a cheap CELL-BE clusters joining a number of PS3s using Ethernet. Such a cluster is still very powerful, but the much slower interconnect may become a problem.

Programming the CELL-BE architecture is very different from programming traditionally homogenous architectures. First of all the two different architectures (PowerPC and SPEs) in the CELL-BE have different properties and two different C/C++ APIs means that the programmer must use two different programming models. Secondly, the programmer must consider how and when the transferring of data between main memory and the LS should happen. Moreover, with a “requirement” of doing transfers and calculations in parallel requires that the programmer incorporates double or triple buffering in the design and in the implementation of algorithms. Finally, the programmer must ensure that the program handles the communication between the PowerPC and SPEs when data transfers start and finishes. In addition, the programs needs some sort of synchronization to make sure that data transfers and data calculations stay synchronized. Programmers are also responsible for implementing this, by sending messages between the SPEs and the PowerPC and/or internally between the SPEs.

All in all, the CELL-BE architecture is very innovative and solves some of the problems with the traditionally Von Neumann based architectures. The downside is that the programming model is very different. With architectures like the CELL-BE the scientist needs to learn parallel programming, a new architecture, programming API, and to setup an environment for developing. Therefore, a number of companies and research groups have proposed systems for simplifying the writing of programs to the CELL-BE. One of them is CellSs[38] tool from the Supercomputing Center in Barcelona and another is Alf by IBM.[39]

The CellSs model is based on code annotations of the parameters to a given method and source-to-source compiling. The programmer marks each parameter as read, write or read/write and then the pre-processor transforms the source code into code which the CELL-BE compiler subsequently can compile. The scheduler identifies data dependencies and data transfers and afterwards it can choose an appropriated SPE for executing the code. Furthermore, CellSs is capable of finding pieces of code or methods that can run on different SPEs in parallel or a least concurrent. The Alf system, developed by IBM, aims to help programmers to handle the architecture by providing methods for creating task and data objects. After the creation of the tasks and data objects, the scheduler of the system handles data transfers including double buffering and load balancing on the SPEs. However, neither systems support a cluster of CELL-BE units and as a result we decided to make a system which did. The next section will describe a distributed shared memory system for the CELL-BE named DSMCBE.

2.2 Distributed Shared Memory for the CELL-BE

As the work done during in our master’s thesis is the basis for the initial work during my PhD project, this section will give a short description of the basic design. In combination, with the published work (see section B1) on DSMCBE, it should provide a good understanding of the system, before the adding of channels and CSP processes.

The s starts with a definition of the target system and an overview of the components creating the DSMCBE system. The target for this system is a cluster consisting of multiple CELL-BE units connected using an interconnect supporting TCP/IP. Furthermore, DSMCBE requires that the programmer firstly identify the data objects begin shared between the units. Secondly, the programmer must ensure that each data object has a unique identifier and finally, makes sure that the data objects have a size that is no larger than the LS of the SPEs.

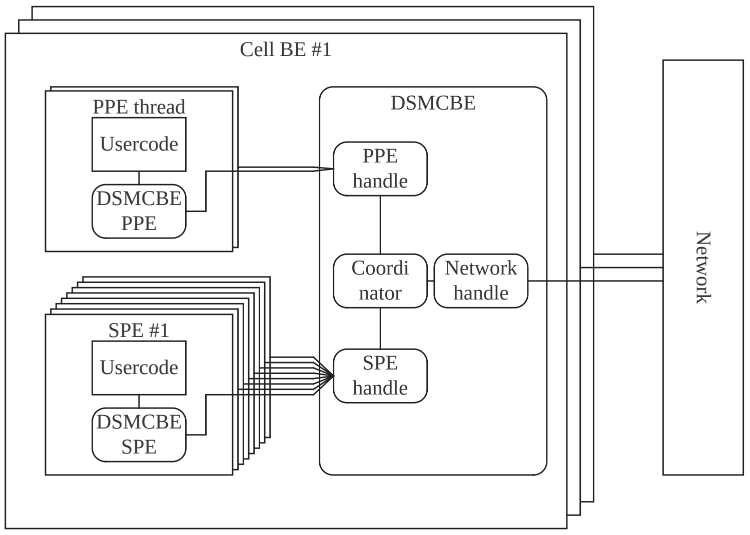


Figure 2.3: The components of DSMCBE and their connections.

Figure 2.3 gives an overview of the DSMCBE design and as seen in the figure the main part of DSMCBE is placed on the PPE unit and consists of the coordinator module with a number of handles attached to it. Each worker (PPE thread or SPE) runs a module containing a number of methods, which the programmer can call in order to use the tool. These modules communicate with the main part using the handles shown in the figure. Furthermore, two CELL-BE machines running DSMCBE can communicate with each other using the network handle. The modules of DSMCBE communicate in different ways. The two modules "DSMCBE SPE" and "SPE handle" communicate using mailbox messages and the network handle communicates with other network handles using Ethernet over TCP/IP. All other modules, which can communicate, do so by injecting messages into the inbox queue of the other module. The system defines a number of methods and packages used for handling the connecting of the units in the cluster, the manipulation of data objects etc.

When having multiple units, the task of finding out which unit has a given data object can be a challenge. However, there exists a number of strategies and one of the simplest is the central server based solution. All data objects are stored on the central server and this makes it simple of all other units to locate data. Clearly, this design only works for small clusters, as the central server very quickly becomes a bottleneck. Therefore, we need another solution, which is able to distribute the data among the units without adding too high an overhead when locating which unit holds a given data object. DSMCBE distributes the data using the simple principle of letting the unit that creates a given data object store the data object. Nevertheless, this idea has some flaws; firstly, how does the other units know, which unit has a given data object. In addition, what if the unit creating the data object only creates the data, but another unit afterwards heavily uses the data object making it preferable to migrated the data object to the unit using it. By combining, the home based distribution model with the idea of each unit having a special data object named “object table” containing a list of all created data objects; all units have a list of which unit created any data object. Furthermore, the units have a starting point for locating a given data object and if the data object has been migrated to another unit, the creating unit will always know which unit the given data object was migrated to. The creating unit will therefore be able to forward the request to the unit that the data object was migrated to and this principle can continue until finding the current owner, which will then directly respond the requester.

At start-up, each unit in the system will have an object table with zero entries. When a unit creates an object, it will send a broadcast to all other units, that it have created an object with a given unique identifier. If two units tries to create an object with the same unique identifier, the system throws an exception.

With data shared among multiple units, consistency is required in order for programs to produce correct execution and results. Chapter 1 briefly described the terms of consistency, so this section will only touch upon consistency in relation to DSMCBE and not in general. Multiple methods for consistency exists, some is very strict and very hard to implement in real life, and others more relaxed. One of the more relaxed models is entry consistency.[17] This model requires programmers to define how the program accesses shared data objects; both in terms of the type of access (read/write), but also when the data objects are no longer used. Below is an example in which a programmer defines how a shared variable is accessed.

```
shared int MySharedDataObject

public Main()
{
    // Call acquire to access the shared variable
    int* value = Acquire(MySharedDataObject, READ-ACCESS)
    print("MySharedDataObject is: " + value)

    // Call require, when the value is no longer used
    Release(MySharedDataObject)

    // The pointer "value" should no be null
}
```

The `READ` and `WRITE` flags defines the mode with which the programmer intents to use the shared variable.

Figure 2.4 illustrates how the home based consistency model works when integrated into the DSMCBE system.

Data objects are always stored on the PowerPC as the small amount of memory in the SPEs LS is unsuitable for storing the data objects including the bookkeeping code for handling the manipulations made to the data objects during execution. The programmer must use the method, shown below, to create a shared data object:

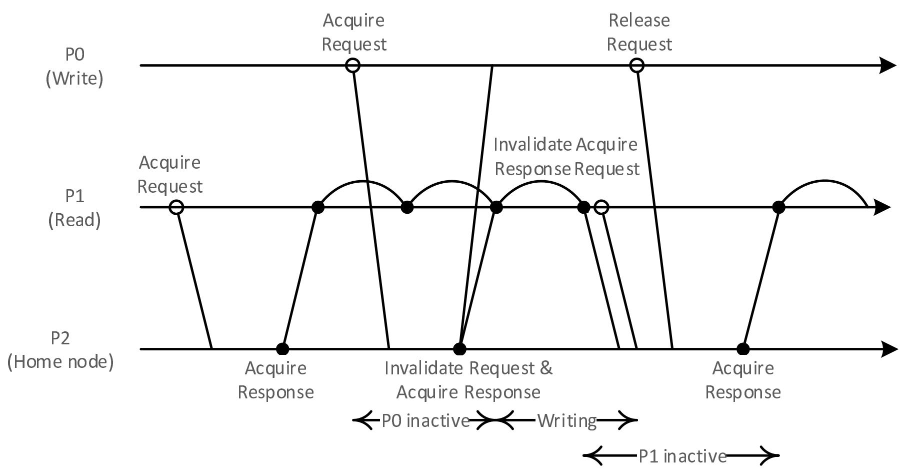


Figure 2.4: Modified entry consistency which can be used when memories are not shared.

```
void* create(GUID id, unsigned long size)
```

The method takes a unique identifier and the size of a data object as arguments and returns the pointer to the data object. Behind the scenes the coordinator responsible for creating data objects check if the object already exists in the object table, if not, the coordinator adds a new entry and informs the other coordinators to update their object tables. Afterwards, the coordinator checks if other units have previously (before the creation of the data object) tried to access it. If this is the case the access request are handle like normal. The reason for this design is to avoid synchronization issues when an algorithm starts executing on multiple machines.

When the programmer wishes to access a data object, the programmer must provide three pieces of information to the method shown below. The first piece of information is the unique identifier of the data object and the second is a pointer to an unsigned long. After the acquire request has been processed, the pointer will contain the size of the data object. Finally, the programmer must define whether access is a read-only or read/write.

```
void* acquire(GUID id, unsigned long* size, int type)
```

The coordinator of the unit, which is the home-node for a given data object, will in most cases handle the access request. As multiple units cannot write to the same data object concurrently, the home-node only grants access to the requestor, if no other units currently have write access to the given data object. Afterwards, the home-node initiates a transfer of the data to the requestor and DSMCBE returns the pointer to the programmer when the data is ready at the PPE or SPE, depending on which unit made the request. In cases where the home-node cannot grant access, the coordinator adds the request to the data objects FIFO queue of waiting request.

A special situation can arise when a unit makes a write access request when others units have previously accessed the given data object. As the SPEs can have a local version of the data object, the system must make sure that all SPEs invalidate and delete any local versions from the LS. This will ensure that data objects maintain consistency. Invalidating a data object is done by the coordinator informing all involved units that they should delete the local version of the given data object. First after all the units respond to the coordinator by sending the invalidate response, the coordinator can allow the write access to be granted to the requestor.

DSMCBE furthermore supports asynchronous acquire requests, meaning that the code can continue working while the coordinator handles, the acquire request and the transfer of the data object the unit. DSMCBE uses a special method to define the synchronization point indicating that the unit cannot execute more code before the data object is available. This allows the programmer to transfer the next data object while doing calculations on the current acquired data object.

When the programmer no longer need access to a data object, she must inform the system by calling the release method shown below:

```
void release(void* data)
```

This will inform the coordinator that the given unit no longer needs access to the data object. In cases where a unit has read access to a data object, the unit can locally handle the release request as the system supports multiple read single write (MRSW). However, the coordinator of the home node must be informed of any releases of write accesses as such accesses will lock and thereby prevent other units from accessing the data object until the unit having access releases the data object.

Implementing a number of algorithms using DSMCBE gave us the possibility to test the performance of the system. The results show that algorithms executing many operations per communication scale very well. However, as the number of operations executed per communication declines, the communication starts to nominate and as a result, performance decrease.

2.3 CSP

The previous section described DSMCBE; a tool for helping the programmer handle the advanced programming model for the CELL-BE. The tool provided the programmer with a number of simple methods for handling shared data objects on a cluster of CELL-BE machines. However, the programmer still needs to be in charge of handling deadlocks, live-locks and race conditions. As described in section 1.1.5 handling these issues is in most cases very advanced and therefore it would be preferable, if the programmer had a tool that by its design remove these issues. In 1978, C.A.R. Hoare[40] defined a formal algebra model that would allow scientists to prove that an algorithm was correct and free of deadlocks, live-locks, and race conditions by splitting up the algorithm into pieces small enough that it would be trivial to argue and prove that the code worked correctly.

This section starts with a small introduction to the practical part of CSP. Thereafter, follows the description of the extensions needed for DSMCBE to support channels and finally, the results and conclusion ends this section.

The CSP model consists of two components namely channels and processes.[40] The model was first introduced in 1978 as a concurrent programming language and in 1985, the process calculus was added to the model.[41]

CSP defines a channel as a named and directed connection between two CSP processes used for explicit message parsing. The idea is that a message is a synchronized event, in the sense that the sender can write to the channel at any time, but has to wait for a receiver to read the channel before being able to continue. Likewise, a receiver can read at any time, but has to wait for a sender to write to the channel. Therefore, both the read and write operations are blocking and the two processes synchronizes execution when communicating. However, in many implementation like JCSP,[42] PyCSP[43] channels with buffering allow asynchronous communication. Some of the common channel types are: one-2-one, any-2-one, one-2-any and any-2-any. The any-2-one means that there exists multiple senders and one receiver whereas the opposite is the case with the one-2-any.

A CSP process is a small unit, which will do a simple computation, such as adding two numberss. A CSP process will take a number of inputs and likewise return a number of outputs. Each input comes from a channel and the process writes outputs into one or more channels. Each process should be simple enough so that the programmer can easily prove the correctness of the code.

Figure 2.5 shows a simple example of a program outputting the Fibonacci numbers using a number of channels and processes. Four different CSP components are used namely PrefixInt, Delta, TailInt and Add. The Prefix component takes an integer as parameter and writes it once to the outgoing channel. Thereafter, the component writes the values read from the incoming channel to the outgoing channel. The Delta component reads the incoming channel and writes the value to the two outgoing channels. The third component (TailInt) reads the incoming channel, discards the value and thereafter just forwards the incoming values to the outgoing channel. The Add component reads the two incoming channels, add the two values, and writes the result to the outgoing channel. This illustrates how to construct algorithms using simple components, in which the correctness is obvious.

2.4 Adding channels to DSMCBE

The first attempt to extent DSMCBE with CSP simply involved a model for supporting channels. The DSMCBE system already supports transferring data between units, making it simple to implement channels as the channels can just use the data existing transfer methods of DSMCBE.

CSP channels are not type-fixed, meaning that it is possible to transfer different sized types e.g. integers and doubles along the same channel. In contrast, DSMCBE forces the programmer to define a fixed size for each shared data object. Therefore, the system must be able to resize data objects, this is done here by extending DSMCBE with the support for deleting data objects. This makes it possible to re-create data objects using the normal create

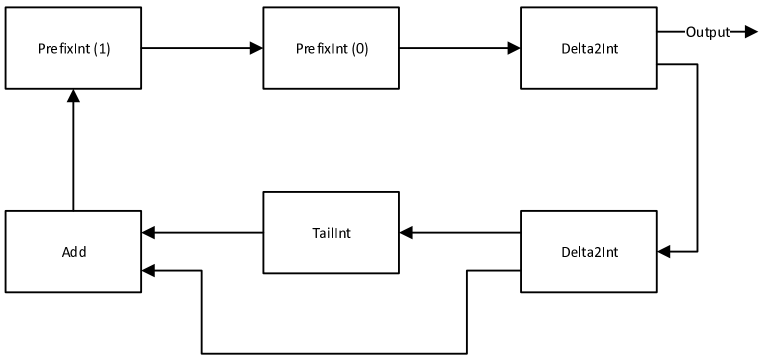


Figure 2.5: A CSP implementation of a Fibonacci generator.

method and thereby define a new size and simulating sending different size values through the same channel.

Finally, DSMCBE must support the synchronized (blocking) behaviour of reading a channel in contrast to the asynchronous read/write methods currently implemented in DSMCBE. DSMCBE already have a method for creating a data object and as it is blocking, there is no reason to change this as it directly maps to the blocking behaviour of writing to a channel.

To make it simple for the programmer to use the system two new methods is added to the API of DSMCBE namely `PUT` and `GET`. The `PUT` is a wrapper for calling the `Create` method of DSMCBE, whereas the `GET` is calling the new asynchronous method DSMCBE method `Acquire-Delete` and then blocking until the result is ready.

We tested the performance of the channel extension in two ways, namely testing the time used on a single channel communication (CommsTime) and testing the overall performance by implementing two benchmarks.

The result of the CommsTime benchmark shows that the implementation in average used 273 microseconds on each communication. This is a bit slower than the time used in JCSP (JDK 1.4),[42] and a factor hundred slower than C++CSP[44] and OCCAM (KRoC 1.3.3).[45] However, the results shows that it definitely is possible to use the system for implementing algorithms and that it should perform as good as other CSP implementations.

The Prototein folding and a Successive-Over-Relaxation (SOR), namely the Heatquation algorithm, was chosen for testing the overall scalability of the system.[46] As expected, the embarrassingly parallel prototein folding scales perfectly as the amount of communication is very limited. In contrast, the SOR requires a lot of communication and as a result it only scales to four SPEs on a single CELL-BE unit.

Overall, implementing channels on top of DSMCBE is definitely doable, but as one could expect the scalability of the implemented algorithms follows the same patterns as seen in the raw DSMCBE version. Furthermore, there are some limitations, as the implementation does not support un-buffered channels, which most channels in CSP are. Therefore, as described in the next section, we chose another approach that also supported CSP processes.

2.5 Adding CSP Processes to DSMCBE

The previous section described how adding CSP to the CELL-BE could be done using a channel model, which resembles CSP channels, on top of an existing DSM system for the CELL-BE architecture. However, the solution did not implement the CSP processes because each SPU can only run a single process, as no threading model exists in the CELL-BE libraries from IBM. Therefore, it is necessary to build a thread model for the SPEs, which will allow each SPE to run multiple CSP processes concurrently.

We start the design by defining the functions, which are made available for the programmer. It will consists of two functions one for creating and one for poisoning (destroying) a channel. In addition, the programmer needs two functions for reading/writing data to a channel and finally, two functions for creating/destroying data areas. The six functions are shown below.

- dsmcbe_csp_channel_create
- dsmcbe_csp_channel_poison
- dsmcbe_csp_channel_read
- dsmcbe_csp_channel_write
- dsmcbe_csp_item_create
- dsmcbe_csp_item_free

In the implementation described in the previous section, every time the program used a channel the result was that the system created, used and then delete the channel. The reason for this design was that data objects in DSMCBE must have a static size. To avoid this behaviour, the new design defines a new object named "transferable item". Such an item is when created ensured to have correct alignment and block size in accordance with the CELL-BE specifications. A pointer is return to the user on creation of such an item, which allow the user to read/write to the allocated data. This allows the programmer to transfer the items over any channel. When a SPE reads from a channel and get a transferable item, it can easy read or modify the data and forward (write) the item to another channel. The design of the transferable item makes it simpler for the programmer, because the design defines a clear separation between data and channels, whereas the previous version had a tight couple between the two.

In all CSP implementations multiple channel types exists for handling multiple scenarios with multiple/single reader(s)/writer(s). Furthermore, some systems allows that a channel has a buffer, which means that the writer can make one or more writes to the channel, before a reader must actually read from the channel. A typical use of such a design is algorithms based on the producer/consumer principle. This implementation defines four types of channels, where each of them can have a buffer which default length is zero. The four types are define below:

- One-2-One
- One-2-Any

- Any-2-One
- Any-2-Any

Finally, a fifth channel type exists, which is a special optimized version of the One-2-One channel. The use of the optimized version named One-2-One-Simple, is situations where the reader and writer always are the same two processes.

The functionality of a CSP process would be rather limited if it could only block on a single channel. Therefore, it is necessary to make it possible to block on multiple channels. Like in other implementations of CSP this must be done using the concept of "external choice". The concept helps avoid race conditions that could easily arise, if each process could peek to see if data was available on a channel and then read the data if available. It is obvious that if another process also peeked at the same time as the first, then both processes would try to read the data, with the result that one of them would get the data whereas the other could fail.

Implementing this is done by making the centralized `coordinator` component of DSMCBE handling the one2one mapping of a reader and a writer. When a process wishes to read multiple channels the `coordinator` receives a single request along with the list of the channels the process wants to read. When another process send a request to write to one of the channels, the id of the channel is match against the pending request and if a mapping exists, the coordinator initiate the data transfer between the two processes. When having multiple processes wanting to read or write to the same channel, the component responsible for doing the mapping must have a strategy for choosing a single process. Looking at some of the other CSP implementations shows, that multiple strategies exists and normally the programmer can choose an appropriated strategy. Among the common strategies are:

Fair The fair tries to even out the number of times each channel is accessed by recording the history of previously accessed channels.

Random The random method just randomly choose one of the available channels with any history.

Priority Using the priority method the programmer defines a priority of the channels, making it simple for the mapper to choose the channel with the highest priority.

The mapper in our implementation use `Priority` method and does not allow the programmer to specify other methods. However, the systems supports the implementation of the other described methods as well as user-defined methods.

A specialized version of the `dsmcbe_csp_channel_read` and `dsmcbe_csp-_channel_write` functions are used namely `dsmcbe_csp_channel_read_alt` and `dsmcbe_csp_channel_read_alt` where `alt` stands for "alternative". The return value of the alternative function is the data from the selected channel along with the unique ID of the selected channel.

When using the alternative methods, the programmer can add a guard to the list of channels to choose from, if she wished an alternative outcome instead of the blocking behaviour, if none of the channels has any data. Normally, two types of guard exists, namely SHIP and TIMEOUT. The SKIP guard simply returns immediately, if none of the channels returns any data, whereas system returns the TIMEOUT guard after a predefined time span. Programmers can also define user functions, which will define the behaviour, if none of the channel returns any data.

In our implementation only the SHIP guard is implemented and the `alternative` functions will return a null pointer and the value of the static SHIP guard.

When creating a new channel it must have a unique identifier, so that the processes can define which channel that they wish to read or write to. In addition, the programmer can specify the buffer-size of the channel. The buffer-size is set to zero, if the programmer does not specify a value. Finally, the user must define the type (One-2-One, One-2-Any etc.) of the channel.

This system uses a FIFO queue for implementing the buffer-size as a fix-sized, whereas the original work on CSP describes the implementation being done using a chain of CSP processes connected by One-2-One channels. Each process will then read a value from the incoming channel and forward the value to the outgoing channel. The length of the chain equals the size of the buffer.

In order to destroy a channel, a poison pill is put into the given channel, which will result in the reader will get the poison and can then accordingly respond e.g. by putting poison pill into all outgoing channels. This will eventually shutdown all channels and terminate execution.

When a process reads a channel, it will receive a pointer to the transferable item. Furthermore, the sending process at the same time turn over the control of the object, meaning the reader is responsible for the data. The reader can modify, forward or delete the data without coordinating with other processes, because the process currently holding the given item has exclusive ownership of that given data.

When building a system on top of a another system like DSMCBE, the starting up of a CSP network of multiple channel and multiple processes can sometimes be error-prone, because the processes start at random and the channels are created at randomly. This could easily result in a situation where a process tries to access a not yet created channel. To avoid such situations, our implementation allows that a process can access not yet created channels. However, the result is that the process will block until some other process creates the channel, afterwards it will process all awaiting accesses in a FIFO manner.

This system will always transfer data via the `coordinator` running on the PPE. The reason for this is due to the design of DSMCBE; however, when using channels some optimization may exist. Firstly, with a system were each SPE will run more than one CSP process, there is a risk of two processes running on the same SPE doing communication through a channel. Instead of first transferring the data through the PPE and then back to the same SPE, it would be simpler just to leave the data on the SPE and then only do the bookkeeping on the `coordinator`. This will naturally increase the performance of the channel communication. Secondly, if two processes were always located on the same two SPEs communication through a one-2-one channel, then it would likewise be more efficiently to transfer the data directly between the two SPEs instead of transferring the data via the main memory (PPE). The programmer can use the optimized channel named `one-2-one-simple` if she can ensure that the two processes always execute on the same two SPEs. Optimally, the system should do all communications without transferring data to the PPE; however, due to the very limited amount of space in the receiving SPEs LS, it is sometimes necessary to transfer the data to main memory while waiting for available space on the LS. Programmer should always try to construct algorithms in a way that will minimize the use of the main memory as cache for the LS's.

With the channel design in place, the time has come to look at the CSP processes. As most CSP algorithms requires a large number of processes, it essential that it is possible to run multiple processes on a single SPE. It would be optimal with an architecture exactly one CSP process per computational unit, but most traditionally architectures only provides in the order of ten or hundred computational units. Therefore, the trick is to define software threads that can in turn be allocate execution time on the hardware. A scheduler is responsible for scheduling the threads in a fair manner to minimize starvation of a single thread.

Natively, there is no support provided by IBM for threading on the SPEs. The reason for this is properly the limited LS and instruction set that together makes threading and scheduling hard and expensive. However, some research have examined how to do threading on the very limited hardware. One of the projects is the CELL-MT[47] that implements a cooperative user-mode thread library.

Like the CELL-MT implementation, this implementation bases on cooperative threading meaning that the programmer is responsible for integrating points in the code where the current thread will yield and thereby allow another thread to do execution. Normally, doing threading using pre-emptive multitasking allows the scheduler of a system to decide when a context switch should occur; however, implementing pre-emptive threading is far too advanced for a small computation unit like the SPE and therefore cooperative threading is chosen.

The design of the thread library changes the way the programmer makes a program for the SPE. The library defines the `main` method of the SPE, meaning that the programmer cannot make their own `main` method. Instead, the programmer must define the `dsmcbe_main` method that will contain the code, which the programmer intended for the normal main method. The reason for this design is that we must place the thread library in the main method to ensure that the system runs the thread library before the programmer's code. The system execute the code inside the `dsmcbe_main` on all of the threads running on the given SPE, meaning that all threads will execute the same main function. However, as the programmer normally wants the threads to execute different code, the library provides a function `dsmcbe_thread_current_id` which will provide the programmer with the current id of executing thread. The programmer can use this ID to determine which code the thread should run. Below is an example of how this could be implemented.

```
void dsmcbe_main()
{
    // Use the current thread ID, to determine with code to execute
    // If thread ID is 1, run the addition and so fort

    switch (dscmbe_thread_current_id)
    {
        case 1:
            do addition()
        case 2:
            do subtraction()
        case 3:
            do multiplication()
    }
}
```

The addition from the example code below, can then contain a `while` loop, that will ensure that the given thread will do additions until the thread is terminated.

The thread library uses the original `main` to implement the functionality of a thread library. When starting the execution of the thread, the library must be aware of the number of threads each SPE should run. The thread library uses this information to allocate space for one stack per thread. The compiler will calculate the maximum size of the stack and write it into the header of the compiled SPE code, so that each thread will get the same stack size. It will possible result in a waste of space, as the thread could potentially have different stack sizes.

With the stacks created, the scheduler can record the pointer to the stack of each of the threads. In addition, the scheduler need the information about the current state (ready or stopped) of each of the thread. Finally, the scheduler record information about the stack pointer, program counter and frame pointer of each of the threads. This information is used by the C functions `setjmp` and `longjmp` that will respectively save and restore the current state of each of the threads. With the information described above, the scheduler is now capable of finding a thread which is ready to run, use the `longjmp` and the stack pointer, to start executing the thread. When the API `dscmbe_thread_yield` method is called, the state of the current thread will be saved using `setjpm` and control will be given back to the scheduler, which will in a round-robin fashion find the next thread ready to execute. In addition to the functions used by the programmer for yielding a thread, the CSP methods for creating, reading, writing and poisoning channels also calls the `dscmeb_thread_yield_ready` which will yield the current thread. This is done to ensure that if the API call blocks due to data transfers, communication with the `coordinator`, then another thread can continue. The programmer can also call `dscmeb_thread_yield_ready` to indicate that if another thread is ready to execute it can do so. This is especially useful when a thread is doing long-lasting calculation without API calls that could result in a yield.

A final remark on this design is that the limited space of the LS naturally limits the number of threads each SPE can run. The implemented test applications, described in the next section, shows that it is difficult to run more than eight threads on a SPE. One could argue that system could transfer the stack and other information about a given thread to main memory and thereby making it possible to run more threads in addition to allowing threads to migrate from one SPE to another. The result would properly a decrease in performance as added overhead of transferring and migrating threads between SPEs would eliminate the gain of having more threads.

Like in the previous section, we will use three simple applications for testing the performance of the implementation. The tests was executed on a CELL-BE blade consisting of two CELL-BE units.

The first test is the CommsTime algorithm, which all CSP implementations use to examine the communication overhead as the algorithms does no computations. Furthermore, CommsTime makes it easy to compare two CSP implementations against each other. The concept of the algorithm is to send a message in a ring a number of time. The time it takes to forward the message divided with the number of communications gives the result.

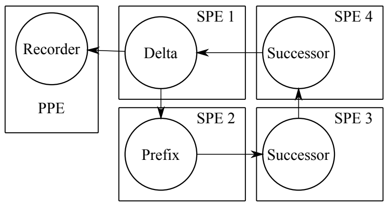


Figure 2.6: Overview of CommsTime.

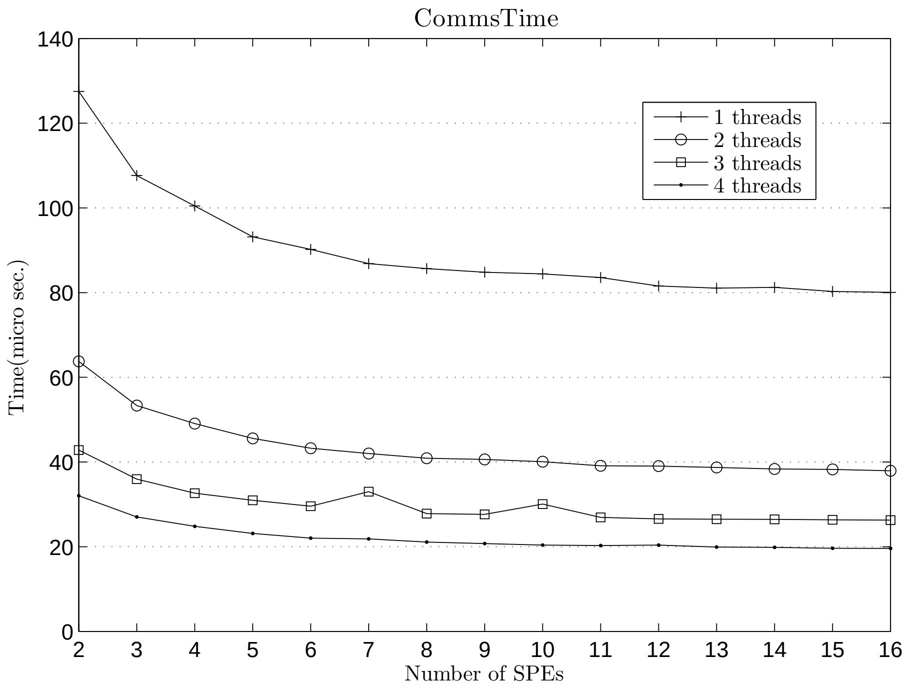


Figure 2.7: The result of running the CommsTime algorithm.

Figure 2.6 shows how we have implemented the algorithm and figure 2.7 shows the result of executing CommsTime using two to sixteen SPEs each running one to four threads. In the most common implementation three CSP processes is used namely a delta, a successor and a prefix. However, in order to use all the SPE's and test the threads, we have inserted more successors to make executions with up to 64 CSP processes possible.

We repeated the test ten times using 10.000 iterations. The results shows two things. Firstly, that the use of threads decrease the time a single communication takes. We expected this, as running more threads on a single SPE increases the amount of internal communication, which is must faster than using SPE communication. Secondly, the tests shows that the communication time decrease when the number of SPEs increases, until a certain point where the communication time stabilizes. The reason for this is that the Delta operation, which include communication with two parties namely the PPE and another SPE, has higher overhead than the Successor operation. Therefore, when the number of processes running the Successor operation increase, the overhead of Delta operation becomes less dominant on the average communication time.

In addition, we tested the CommsTime algorithm running on a PPE without using the SPEs in order to compare the performance again another CSP implementation namely JCSP. It turns out the communication time of JCSP is a least twice as low as our implementation running on the PPE. When we using more than six SPEs, our implementation is faster when using more than one thread per SPE.

The second test is a prototein[48] folding algorithm and the algorithm is implemented using the bag-of-task strategy, which is illustrated in figure 2.8. As one could imagine, knowing that the algorithm is embarrassingly parallel, it scales perfectly for all runs (see figure 2.9).

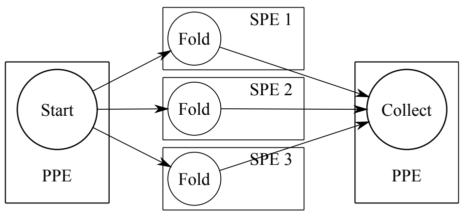


Figure 2.8: Overview of the Prototein implementation.

The final application is the K-nearest-neighbours (kNN) simulation adopted from the PyCSP[43] library. Previous in this section we described that this library will only allow a small number of threads (processes) per SPE. When executing the algorithm it is possible to adjust the problem size by changing the number of elements and dimensions of each element. As seen later on, these adjustments plays a central role in getting good performance, but consequently the higher number of elements and dimensions also minimizes the number of threads per SPE.

Finding the ten nearest neighbours in a setup with 50.000 elements each with 75 dimensions resulted in figure 2.10. The figure illustrates that there is a linear scaling; however, the gradient is only 0,85x, meaning that the library performs 85% of what would be the optimal. Furthermore, using two threads only improve performance a bit; one would expect a bigger improvement, as the two threads would indirectly implement double buffering meaning that

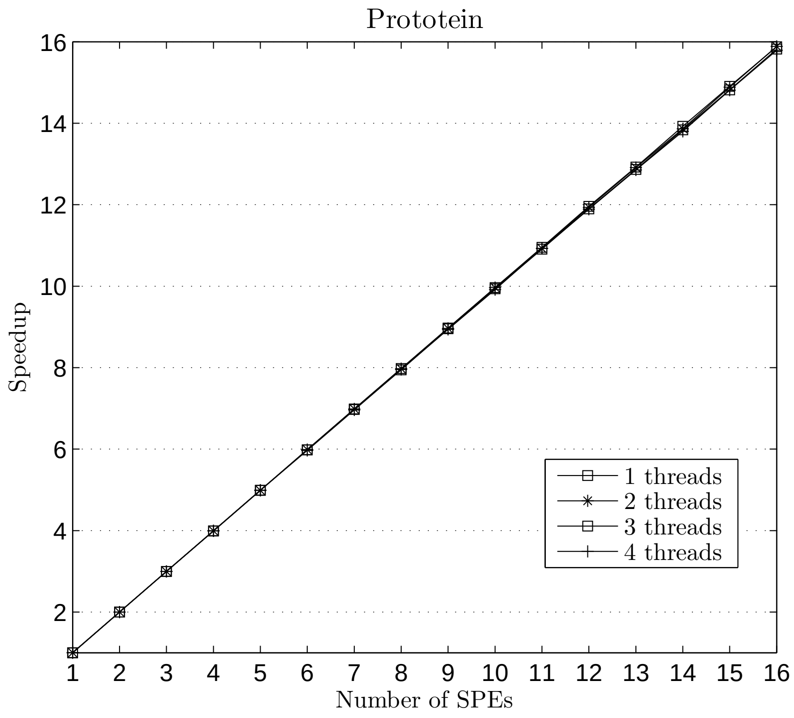


Figure 2.9: The result of running the Prototein algorithm.

while one thread was communicating and transferring data, the other thread could do calculation. Overall, the result is satisfying as it is normally difficult for libraries like this to achieve good scaling when running these n^2 problems especially on architectures with small memories. The reason for this is that the amount of calculations done compared to the amount of communication is very small.

To show this phenomenon, a small algorithm was implemented that have the same communication properties as the n^2 problems, but the amount of computation per communication is set as a parameter. However, the implementation allows no internal SPE communication (see figure 2.11) in order to be able to know the exact number of communications. In addition, the amount of data transferred is in each communication static (16Kbit), so that the only variable is the amount of floating point operations done. Finally, the amount of packages (tasks) was static and each package contained information about the amount of floating-point calculations that each thread should execute. We chose different values for the total number of floating-point operations and repeated the tests ten times to ensure stabile numbers with one and two threads per SPE. Figure 2.12 shows the results of the executions, and it is seen that to get good scalability, a total of circa 100 Mflops is required, less than that will result in poor scalability. In addition, one can see, that using one or two threads per SPE does only add extra overhead, meaning that the latency is so low, that the time used on thread switching is much higher than the latency.

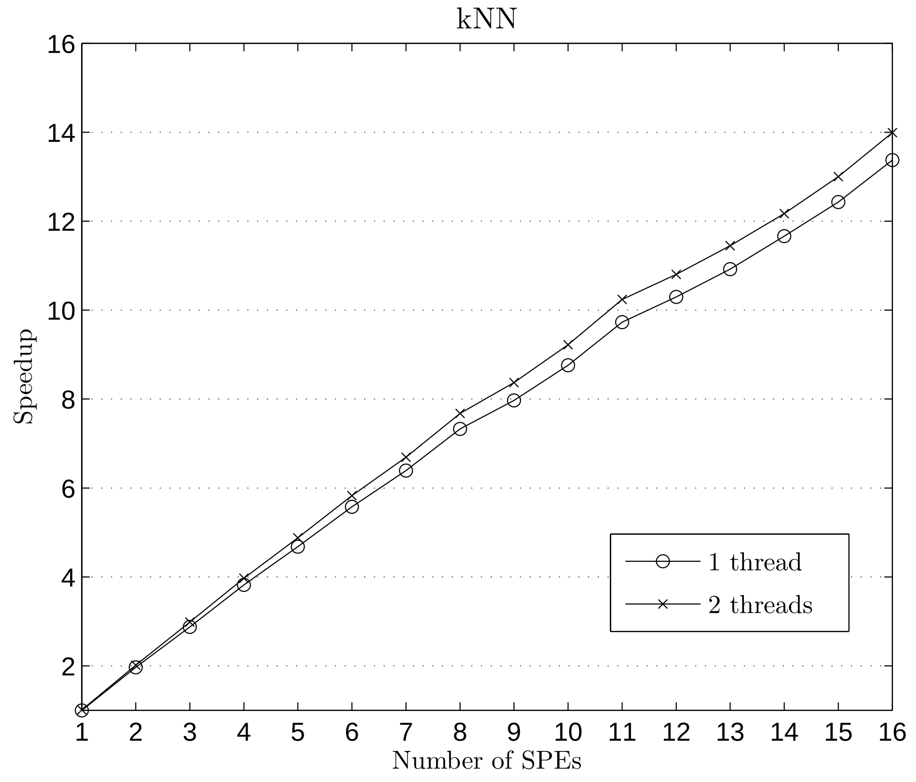


Figure 2.10: The result of running the kNN algorithm.

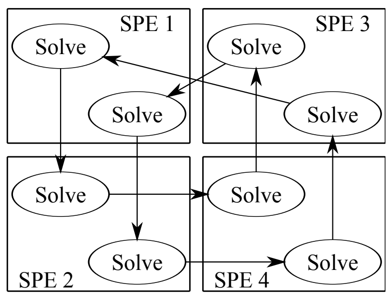


Figure 2.11: Overview of the Communnication-To-Calculation implementation.

However, this does not mean that the latency is as low as possible, but only that it is lower than the context switching of threads on the SPEs.

2.6 Conclusion and perspectivation

The second attempt to implement CSP to the CELL-BE was successful. Based on the implemented algorithms and the usability of the library, it is definitely faster and easier to implement algorithms than the first attempt. Furthermore,

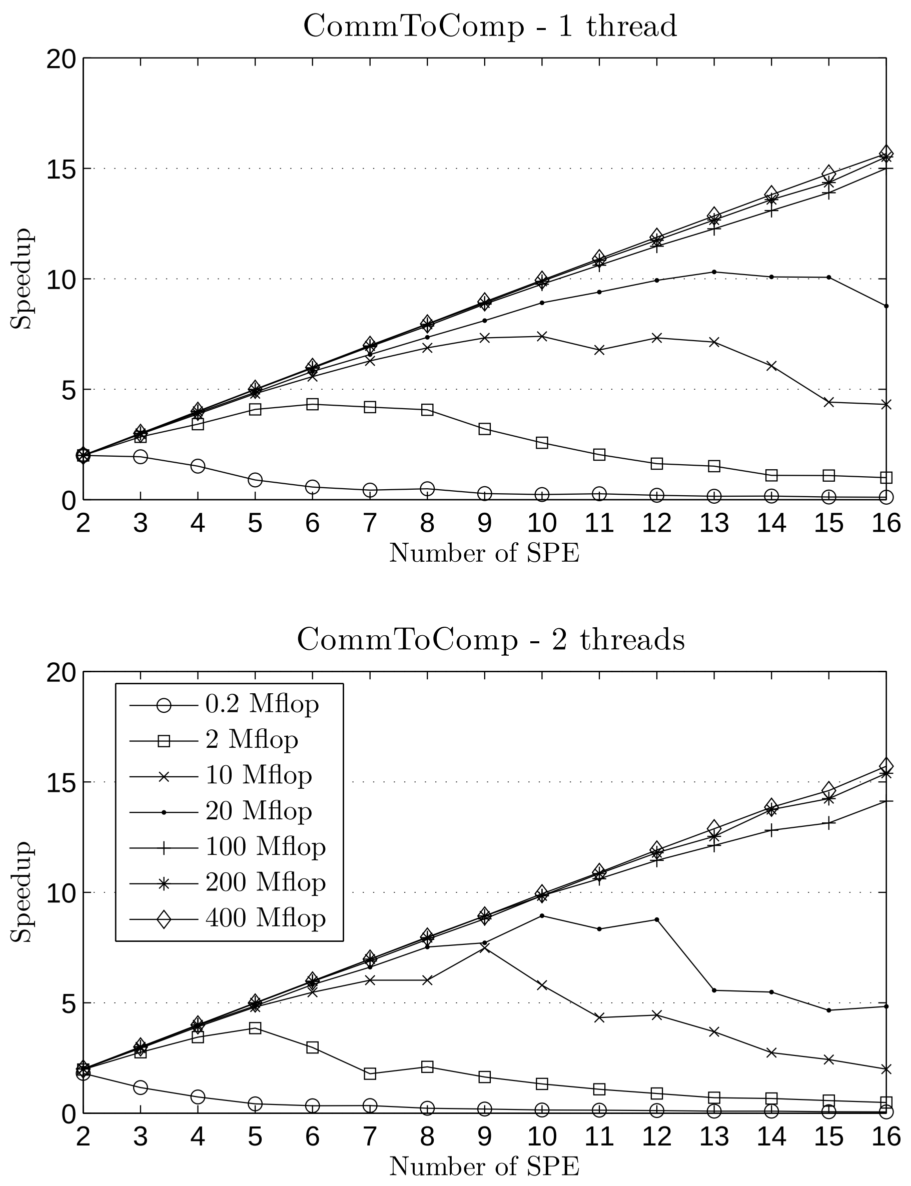


Figure 2.12: The result of running the Communnication-To-Calculation algorithm.

the addition of CSP processes makes the library more complete. However, both libraries suffers from the performance related problems with the DSMCBE system, namely, that the price of doing communication requires that algorithms must have a high computation/communication ratio to get good scalability.

The tests done on the libraries was all done on single CELL-BE blade consisting of two units. The system supports a network, but the price of communicating over Ethernet is too high to get good performance, unless embarrassingly parallel algorithms are tested.

If someone wanted to re-implement the this library, the DSMCBE system should properly be broken down to the components needed for making a CSP system and unused components should be removed. Then it would be easier to concentrate on optimizing for CSP, without at the same time ensuring that the DSM system also would work. Furthermore, more SPE-to-SPE communication would greatly improve performance, as it would allow executing parallel transfers. Likewise, thread migration could be a possibility, which would allow moving the program to the data instead of doing it the other way around.

Chapter 3

Distributed Virtual Execution System

The previous two sections described work with the CELL-BE as the target architecture and even though the architecture is interesting IBM had already decided to discontinue work on it when I started on this PhD project. In combination, with a desire to explore another architecture, I decided to focus on something more common namely the x86/64 architecture. However, the idea for making it easier for the programmer to write code still applies. Even though the use of Microsoft .NET as the target programming model is not widely used in high performance computing, some of the programming languages supported by .NET like IronPython and F# could very well be used for writing high performance applications. As .NET compiles code written using any of the .NET programming languages to the same intermediate code named Common Intermediate Language (CIL), before being Just-In-Time compiled on the target architecture, the platform seems interesting to explore.

This chapter will start with a small introduction to the x86 architecture and the following sections will describe a system name DistVES that is a virtual execution system for executing CIL code on a small distributed system. The essential work is the distributed object systems and the code generator used for a CIL-to-CIL compilation in order for the original program to support the distributed object system.

3.1 x86 Architecture

For many years the x86 architecture has dominated as the most used architecture in personal computers, but it is also highly used in large cluster computers. However, it is Von Neumann based meaning that there is a limit of the maximum clock frequency of the computational units due to issues with heating (power wall) and pipelining (frequency wall), but also issues with the memory and bus being much slower than the computational units (memory wall).[49] The latter, makes it hard to provide the computational units with enough data from memory to keep them working at maximum speed. In an attempt to overcome all of these problems, engineers lowered the clock frequency and they used the extra transistors on making multiple computational units on the same socket (multi-core processing). As, described in chapter 1, this made it even harder for scientists to write small, but compute intensive programs for executing on their personal machines.

The hardware in the modern versions of the x86 architectures have support for doing vector operations meaning doing the same operation on multiple data in parallel. One example could be the multiplication of each integer in a list with a scalar. Normally, programmers could write this with a `For` loop and the result is a sequential run with one multiplication per integer in the list. However, the architecture supports doing this operation in parallel, that would allow four multiplication in parallel using 32bit integers and 128 bit registers. This means that the number of multiplication executed can be reduce with a factor of four, thereby reduce the total execution time. The term Streaming SIMD Extension (SSE) defines this principle and many programming languages support the Single-Instruction-Multiple-Data (SIMD) or vector operations that will result in using the special SIMD instructions from the x86 instruction-set. However, the level of parallelism is limited when using these instructions and normally one would want parallelism on a higher level than on the instruction level as described in chapter 1. Furthermore, using the SIMD instruction only gives parallelism within a single x86 processor, as instruction level parallelism is limited to a single machine.

Therefore, programmers must use another source to achieve high parallelism and scalability. For many years, they have done this by writing programs that use a high number of concurrent threads and then letting the operating system's scheduler handle to job of making the threads run in parallel. But, many programmers finds it very hard to write programs, that is capable of executing a high number of threads and especially doing this in a way that will ensure that the implemented algorithm will produce a correct result. Therefore, writing parallel program for the x86 architectures can be done in other ways and some of them like OpenMP, MPI etc. are already described in chapter 1. However, Microsoft has also made support for writing parallel program for the x86 architecture using .NET with their Task Parallel Library. Nevertheless, the library only supports a single machine, and the project described below makes another approach to achieve parallelism on multiple machines based on Microsoft's idea.

3.2 Design

The first section describes the overall design of the system, and the following sections describes some of the components in more details. The design is inspired by the TPL library and is Parallel.For construct. As the name indicates, the Parallel.For is a parallel version of a For-loop where the library execute the body in parallel utilizing all available cores on the executing shared memory machine. This design makes it very simple for the programmer to write parallel code as the programmer does not need to worry about threads etc., but it does not relieve the programmer from using time to ensure that the program does not contain race conditions, dead locks and so on.

The DistVES version of the loop named `DistVES.Parallel.For` takes three parameters namely the two loop variables (`From` and `To`) and then an Action<int> which is a delegate that takes a single integer as input indicating the current value of the loop counter. The rest of the system bases on a centralized server design with a number of TCP/IP connecting clients, which each can executed a single task at a time. We chose a central server design even though it sets a limit in the scalability. However, one can always change it later to use a distributed server or home node based design. Furthermore, as we plan to use a maximum of 16 clients, the central server design should be able to show if there is potential in the proposed idea. Each client has a single thread that is responsible for handling the communication with the server and the result is that clients can simultaneous communicate with the server and do calculations. To utilize all cores on a single machine, the system allows the running of multiple clients on a single machine, but two or more client on the same machine will not share any information or memory. The alternative is to only allow one client per machine, but allow this client to execute multiple threads. This could minimize the amount of communication between machines running clients and the server, as the system could allow all threads started by a client, to access all the shared objects on the client in a shared memory manner. However, the current version of DistVES does not include this extension.

The user starts the server by providing it with a compiled .NET program with one or more `DistVES.Parallel.For` constructs as argument. The server will then modify the code using a code generator which is described in section 3.4. When the code generation has successfully completed, the system generates a task containing the `Main` method of the program and sends it to one of the available clients. The chosen client will then execute the code of the main loop and at some point meet the `DistVES.Parallel.For` constructs. At this point, it will call DistVES code, which will execute the loop and generate a single task for each loop iteration containing the Action<int> and the current value of the loop counter. The client will send the tasks to the server, which will distributed the task(s) to the connected clients. The client running the main loop, will after the task generation block until all tasks has completed, and will not be able to run other tasks. With all tasks done, the result is send to the blocking client and the main task will continue execution.

The server is responsible for handling the distribution of task (scheduling) and it is done using a simple scheme were the server holds a list of current clients and their current state (waiting/running), if tasks exists and there are client with state "waiting" the clients will be assigned a task as long as tasks exists. If all clients are running, the server will wait until a client will become ready.

The next sections will describe the most essential components of the system.

3.3 Consistency

Consistency is one of the most central components when making a system for distributed shared memory, as the consistency model is the model that will define the guarantees about the state of the same memory region located on different interconnected machines. The optimal solution would clearly be that all memory regions on all machines would stay synchronized at all time. The term Strict Consistency defines this idea and is in reality impossible to implement in any distributed system as such systems always have latency between the memories on the different machines.[50] This means that two machines will never see an update to a memory location at the exactly same time, and therefore reading the same memory region on two machines will at some point yield different results. Consequently, we need more relaxed consistency models in order to build distributed shared memory systems. A number of models exists each with different properties, but they can be divided into two categories namely the models that requires that the programmer defines synchronization points and the models the does not. Generally, the models the use some sort of synchronization are more relaxed than the models that does not, but they also requires more work from the programmer and cannot be used for hardware circuits like caches. The advantages of using a more relaxed consistency model is that it will be able to scale linear also in larger systems. The reason for this is the time span available for doing the synchronization of memory regions across multiple machines increases as the relaxation of the consistency model increases. However, there is also a penalty, as the more relaxed models requires more information about the access patterns of memory regions. One of the most efficient consistency models is "Entry Consistency",[17] but the model requires information on when and how a program accesses any given memory region and furthermore when the program is done using the region. Programmers will use the `Acquire` and `Release` method to indicate this. Other models like the less relaxed "Release Consistency"[51] model, requires only the use of a single synchronization flag, which the programmer use to indicate that all memory regions should be synchronized at the given point.

As the goal of the design proposed is to make it simpler for the programmer to write programs that will run in parallel on multiple machines, the more relaxed consistency models with their requirements of the programmer indicating synchronization or access patterns is incompatible with this design goal. Therefore, we must chose a model like Sequential Consistency, which is less relaxed, but does not require explicit synchronization points. Lamport[52] defines the sequential consistency as follows:

```
A multiprocessor system is sequentially consistent if the result
of any execution is the same as if the operations of all the pro-
cessors were executed in some sequential order, and the operations
of each individual processor appears in this sequence in the order
specified by its program.
```

The definition consists of two statements. Firstly, if a process updates the value of A and then two other processes simultaneous (exact same time) access A then they will both either see the updated value or not. After one process has seen the updated value, then all other processes will also see the updated value if they access it. Figure 3.1 illustrates a valid run under sequential consistency, but figure 3.2 shows a run where the visibility clause is violated.

Secondly, when a single process updates both A and B, then all other processes must, when one process have obtained the updated value of B, also obtain the updated value of A if accessed. See figure 3.3 which illustrate an invalid run where the program order is violated.

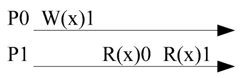


Figure 3.1: Valid under sequential consistency, but not using strict consistency.

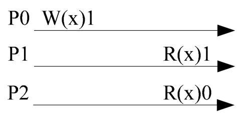


Figure 3.2: Sequential consistency where the visibility clause is violated.

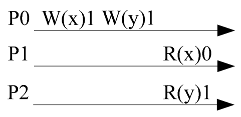


Figure 3.3: Sequential consistency where the program order is violated.

However, Lipton and Sandberg[53] showed that there is some problems with the model, specifically that an optimization of the read or write latency will worsen the latency of the other. Therefore, this consistency model is not perfect, but with the need for specifying access patterns, the model fits well with our design goal. Even though the model is not perfect, it has been implemented in the MESI protocol[54] used in low latency caches like the caches in the Intel Pentium processors.[55] Sequential consistency is in DistVES implemented by using MESI protocol as the following section describes.

The MESI operates with four different states as seen below:

Modified The current data object is the only valid in the system and other machines still have invalid versions of the data object.

Exclusive Like modified; however, other machines may NOT have an invalid version of the data object.

Shared The current data object is valid and may exist on multiple machines.

Invalid The current data object is invalid and an updated version must be requested on next access.

When a process access a shared data object the outcome will depend on the state of the given data object and the state of an object can be update in two ways i.e. active and passive cache actions. An active cache action happens when the code running on a process access a shared object placed in the local memory. Whereas a passive cache action happens when another process is accessing a shared object and that access result in some kind of communication with the server, which in the end results in the change of the state of the shared object on all processes. As all communication in DistVES goes through the centralized server, a process doing an active cache action on a shared object, the process will only inform the server of its intents, and then the server will communicate or “snoop” as the MESI protocol defines it, the other processes which have a copy of the given object. Figure 3.4 shows how the state of a shared object placed in the memory of a given process will change when access locally or another process access a copy of the same object.

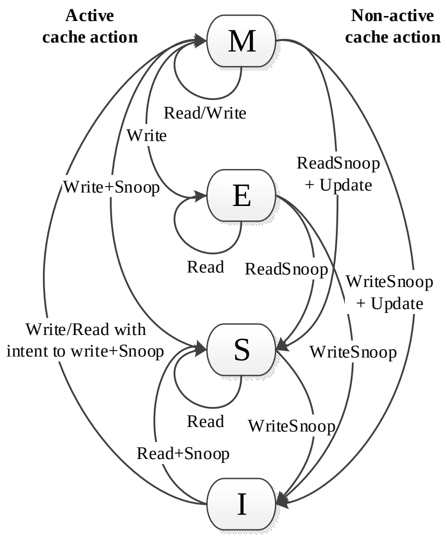


Figure 3.4: Illustration showing the state transitions in the MESI protocol.

With the MESI protocol in place, one must find a way to incorporate the MESI protocol in the fields shared by multiple machines. One solution is to call a special method each time a program access a shared field. The program should call this special method before a read and after a write access, and the method must then do the necessary work based on the state of the shared object and according to the rules defined by the MESI protocol. The method would need information about the accessing instance (class object), the field being accessed and the type of access (read or write). The instructions needed to do this are ldarg.0 (load `this` pointer), ldstr (load the name of the field) and then load an integer indicating “read” or “write”, and finally the method must be called. This approach is simple, but also have some problems. This requires extra work to handle array element access, but this could easily be handle by defining an extra parameter to the special method to indicate the index of the accessed element in the array. However, the problem occurs when dealing with fields containing references to an object, which multiple machines share. Tracking such an object is not simple and if one forgot to call the special method, the whole system would fail. Therefore, to ensure correct update of such objects we propose to change the type of fields and shared objects. This will still require the code generator to track referenced objects, but it will remove the risk of running a program with unmodified accesses to a shared object as .NET being type safe. Only if the code generator modify all accesses to a given shared object correctly will the type system be satisfied. Furthermore, the same idea of changing the type of shared item are used on the shared fields them self. The downside of this approach is that it requires more work, as many different scenarios must be handle as a reference field can be single or multidimensional arrays, user defined types or system types like strings. Some types pose a problem as they are not marked as `serializable` and therefore cannot be serialized and thereby transferred between machines over network. Nevertheless, in this initial version it will be the programmer's responsibility to ensure that, all shared fields must be of the serializable type otherwise the system will fail.

As previously stated there exists two types of fields in .NET i.e. value-type and reference-type. If the field is value-type like integer, float, double etc. the value will be stored in the field, whereas a reference-type field will contain a reference to an object. With the value-typed field the value is read/written directly to the field, but with the reference-typed field the pointer to an object will also be read/written, but the most used situation is that the pointer is loaded and then the object which the pointer points to will be modified. As both the updates made to the pointer and the updates made to the object must be shared along the machines, the MESI protocol must be implemented in both the reference-field and the referenced objects. To distinguish between value-typed and reference-typed MESI fields, we define two types of MESI field namely `MESIValueField` and `MESIReferenceField`. The `MESIValueField` (see figure 3.5) will have the type of the original field as the generic parameter. The `MESIReferenceField` (see figure 3.6) will also have a generic parameter, but it will be another MESI type namely the `MESIReferenceType` which is a special construct used for handling pointers (references) in a safe manner. The thought behind this construct is that the reference-typed field will point to some data. The MESI protocol must ensure that the data remains consistent, whereas the field itself only contains a pointer that the programmer can update from time to time. Therefore, the `MESIReferenceType` contains the MESI protocol and the `MESIReferenceField` must contain logic, which handles the case where the pointer is updated with a new pointer. The system must distribute all such updates among all clients, so that the system remains consistent. A special version of the `MESIReferenceType` is the `MESIValueArray` (see figure 3.7) which as the name indicates handles value-typed arrays. The reason for this special type is that the nature of arrays are very different from other references especially the access of array elements and the special CIL instruction used to get the length of an array. In addition, when working with arrays it will sometimes be best to have the MESI protocol implemented on each element and at other times, one MESI state for all elements are preferable. The first case will produce more communication; however, having one MESI state for all elements will result in a long waiting time as only one client can access the array at a time, and therefore the implementation should be able to detect at runtime which granularity is the best. A simple way of detecting this is to measure the time between accesses and if many clients accesses elements very frequently the system could increase the granularity. However, in the current implementation the programmer must specify a static granularity during the code generation.

An array of arrays or jagged array, e.g. `int[][]`, is like the single dimension value-array encapsulated inside a `MESIReferenceField`, but instead of using the `MESIValueArray` it is a `MESIReferenceArray` containing a number of `MESIValueArrays` (see figure 3.7).

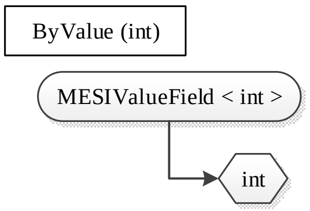


Figure 3.5: MESI container for value types.

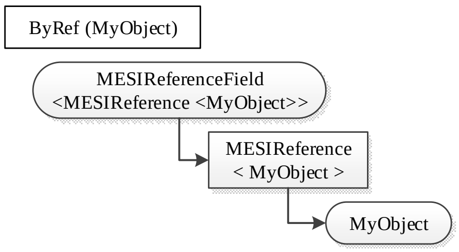

ByRef (MyObject)
MESIReferenceField <MESIReference <MyObject>>
MESIReference < MyObject >
MyObject

Figure 3.6: MESI container for referenced types.

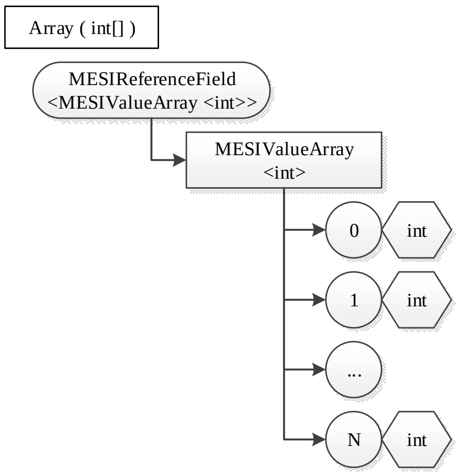


Figure 3.7: MESI container for array.

3.4 Code Generator

The second step is to design a mechanism (a code generator) for transforming the compiled user code into compiled code that will support the design described so far. The goal of the code generator is to identify the fields that the program accesses from multiple machines and then modify the code accordantly to obtain consistency like described in the previous section. However, first we must identify which information we can deduce from the CIL code. The source code below is an example of a program with a `Parallel.For` loop and below is the assembly view of the compiled code with the CIL instruction omitted.

```
class Test
{
    // The following variable is a private field in the Test class
    int[] globalValue;

    public void Run()
    {
        // The following local variable in the class Test is "promoted"
        // to a field in the delegate because it is accessed within the
             delegate
        int value = 4;

        // Initiate the (private) class field.
        globalValue = new int[10];

        /*
         * Code inside the DistVES.Paralle.For loop is compiled to a subclass
              of
         * Test (a delegate). If the body of the Parallel.For did not touch
         * local variables in the Run method, the body of Paralle.For would
         * be compiled to a method in the Test class
         */
        DistVES.Parallel.For(0, globalValue.Length, i =>
            {
                int count = i * value;
                globalValue[i] = count;
            }
        );

        // Next line gives compiler error, because the variable count is out
             of scope
        // value = count;


        // Print the result
        StringBuilder result = new StringBuilder();
        foreach(int integer in globalValue)
            result.Append(string.Format("{0} ", integer));

        Console.WriteLine("Result is [{0}]", result.ToString().Trim());

        // Wait for user input to quit program
        Console.WriteLine("Press any key to continue!");
        Console.ReadKey();
    }
```

```
___[MOD] ...CodeExample.exe
   |     M A N I F E S T
   |___[NAMESPACE] CodeExample
   |   |___[CLASS] CodeExample.Test
   |   |   |     .class private auto ansi beforefieldinit
   |   |   |___[CLASS] <>c__DisplayClass1
   |   |   |   |     .class nested private auto ansi sealed beforefieldinit
   |   |   |   |     .custom instance void [mscorlib]System.Runtime...
   |   |   |   |___[FIELD] <>4__this : public class CodeExample.Test
   |   |   |   |___[FIELD] value : public int32
   |   |   |   |___[METHOD] .ctor : void()
   |   |   |   |     <Run>b__0 : void(int32)
   |   |   |
   |   |   |___[FIELD] globalValue : private int32
   |   |   |___[METHOD] .ctor : void()
   |   |   |___[METHOD] Run : void()
```

The first thing to notice is that the body of the loop, when compiled, turns into a subclass (`<>c__DisplayClass1`) of the original class. This subclass functions as an "delegate", which is is a kind of type safe method pointer. The second thing to notice is that the delegate has two fields, one being a reference to the parent class and the second is the shared variable `value`. Finally, the class contains a constructor and the `Run` method containing the actual code of loop.

With the information shown above and the CIL instructions (omitted in the figure), the code generator must first find the code which will run in parallel. As the programmer has identified these pieces of code by calling the `DistVES.Parallel.For`, the code generator can simply search through all the `CALL` CIL instructions and see if the `DistVES.Parallel.For` method is the operand of the instruction. When the code generator finds a parallel method, it must back trace its parameters and thereby locate the creation of the Action. Then the code generator must do another back trace to find the body of the Action<int> which equals the function the programmer wishes to execute in parallel. The code snippet below illustrates this based on using code from above.

```
...
...
IL_002c: ldftn  instance void CodeExample.Test/'<>c__DisplayClass1'::'<Run>b__0
    '(int32)
IL_0032: newobj instance void class [mscorlib]System.Action'1<int32>::.ctor(
    object, native int)
IL_0037: call   void [Tasks]DistVES.Parallel::For(int32, int32, class [mscorlib]
    System.Action'1<int32>)
...
...
```

The code generator will identify instruction `IL_0037` as the Parallel loop and then the back trace and finally identify line `IL_002c` as the body of the loop. Now, the `<Run>b__0` method in the DisplayClass1, has been identified as the body and the code generator must save this information in order to continue the search for other loops.

When the search has finished the code generator must further analyse the found "parallel" methods in order to identify the fields which will be shared by multiple machines. With the method identified, identifying the fields accessed within the method is a simple process of examining all the CIL instructions in a given method and noting all fields that are accessed using the `LDFLD` or/and `STFLD` instructions. However, caution should be exercised as the method examined could call other methods and in this method access fields, which could potential be shared between multiple machines. Nevertheless, the fix is simple as one just have to recursively find `CALL` instructions and notate the called methods. This is not a watertight solution as the methods called could be system methods, which the code generator cannot directly examine. On the other hand, most system methods will rarely contain shared fields and therefore, in this version of DistVES the code generator will skip calls to system methods.

With all the fields identified, the next step is to convert them to one of the appropriated MESI types described in the previous section. With the fields converted to a MESI type, the code generator must find and convert all load field and store field instructions. Both load and store field operations will, in the converted code, start by loading the field which was converted to a MESI type, and then either the `Get` or `Set` property of the field is called depending on whether the original operation was a load or store.

With reference-typed fields, this can be a bit trickier as referenced object can be loaded from a field and then passed on to another method as parameter. But, as the field is now a MESI object the type of the parameter must be changed, which again may lead to changes of the type of local variables, other method parameters etc. This approach is unfortunately not possible with system methods and therefore these situations need special handling. A way to handle this situation is to lock and read the MESI object before passing the original object to the system method. After execution returns from the system method, the system must write to the MESI object to ensure that an eventual write made to the original object in the system method is propagated to the system. This is not a perfect solution, but with the design where DistVES is not integrated into the VES of .NET there exists no other ways of doing it.

In .NET, the invocation of non-static methods includes a target, namely an instantiated object of the class that the method belongs to, and therefore all clients must have a copy of instantiated class. However, as we cannot guarantee that the .NET can serialize the class at runtime, because it may not have the serializable attribute, we must find another way. One way is to send the CIL code and then let all clients create their own instantiated version of the class. However, the original constructor contains code to initialize the shared fields, but to ensure correctness of the code one must only initialize the shared fields once. Therefore, we want the client that executes the main method to run and thereby initialize the shared fields. Therefore, we must add another constructor to the code to ensure that all other clients can also create a valid initiated class. Using the DSM system, this added constructor must contain code to load (read), instead of create, the values of the shared fields when creating the class.

3.5 Results

To demonstrate the potential in DistVES we implemented three benchmarks and executed them using both DistVES and the TPL library in order to compare the two solutions. Which library to use was indicated using a simple switch and this ensured that the source code, besides the call to the `Parallel.For` loop, is the same no matter what. We executed the benchmarks on a system consisting of four machines connected through a gigabit network. Each machine was equipped with an Intel i7-860 quad-core processor running at 2.80 GHz and with minimum 8GB of memory. As the Microsoft implementation only supports a single machine, we also executed DistVES in a version running only on a single machine, to get a better comparison between the two methods. In addition DistVES was executed using two network setups; firstly, a setup named "Network (3)" with three machines running the same amount of clients and the fourth machine running the server. In the second setup, named "Network (3+1)", all machines ran the same amount of clients and one of them was in addition running the server.

The three benchmarks chosen all belongs to the same category of algorithms that should yield linear speedup. However, they all have different properties like the use of synchronization mechanisms, size of input/output data, data access patterns etc. We executed each benchmark five times and the average of the execution times were used in the figures shown below.

3.5.1 Prototein

The first benchmarks is the "Prototein folding" algorithm already described in the previous chapters. To summarize it is an embarrassingly parallel algorithm with very little communication between the clients and the server. Furthermore, the amount of data used in this algorithm is low and in combination with a good calculation/communication ratio, we expect linear scaling with a gradient close to one.

Figure 3.8 shows the result of the execution, and the overall impression is fairly linear scaling up to two clients per machine, but the gradient is only around 0.8, meaning that there is a potential overhead of 20 %. Furthermore, the graph shows that the single machine version of DistVES is slower than the three other executions. The reason for this is properly that same as the reason for the scaling decreases when running more than two clients per machine in the network case, namely the added overhead of handling accesses to array elements. Compared to the single instruction for accessing an element in the .NET, the DistVES version requires around 5-10 instructions in order to maintain consistency in the simple case were the accessed element already is available to the client. In the network cases the decrease of the scaling is also related to the machine running the server being overloaded, as seen in the graph, the decrease of scalability when running "Network (3)" is less compared to running "Network (3+1)".

3.5.2 Black-Scholes

The Black-Scholes is use in the financial industry to estimate the price of European style options. The algorithm use a minimum of data and generates a single value as output. The calculation/communication ratio is very high and therefore good speedup is expected; however, based on the outcome of the previous benchmark, we expect a decrease in scalability when running more than two clients per machine.

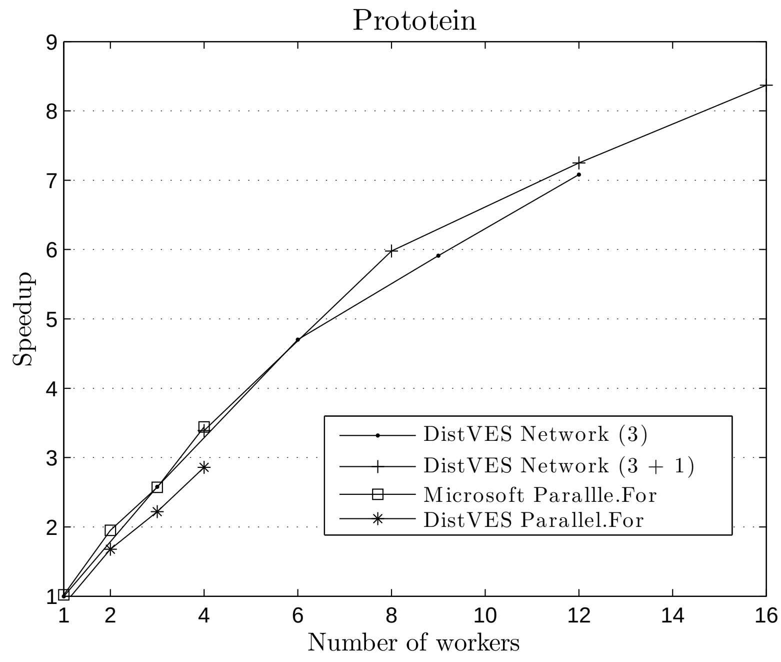


Figure 3.8: Prototein benchmark

The graph shown in figure 3.9 is almost a copy of the graph for the Prototein folding benchmark, but the gradient decreases a bit faster when using more than two clients per machine in the network executions. The reason for this decreased is that an increase in server load when using more clients, which result in longer processing time and therefore the clients will experience a longer waiting time between sending a request to the server and getting the result. We measured the time span between task creation on the server and the start of task execution on the clients. This time span increases when using more clients and when running with 16 clients the time span accounts for 25 % of the total execution time. Overall, the result is acceptable especially taking into account that the code is easier to write than code using threads.

3.5.3 Ising

The last experiment is the Monte Carlo based Ising simulation, which is a mathematical model for simulating magnetism in statistical mechanics. Like the previous algorithms, it is embarrassingly parallel, but unlike the previous algorithms, this use a barrier to synchronize the rounds in the simulation. Therefore, we executed the test with a dynamic problems size unlike the previous two benchmarks, which both used a fixed problem size. The problems size is increased with the number of clients meaning that if one client has problem size x, the problem size of n clients is nx. The Gustavson graph is used to illustrate this and here optimal scalability is illustrated by a horizontal line instead of a linear line with gradient of one. Even though the benchmark is

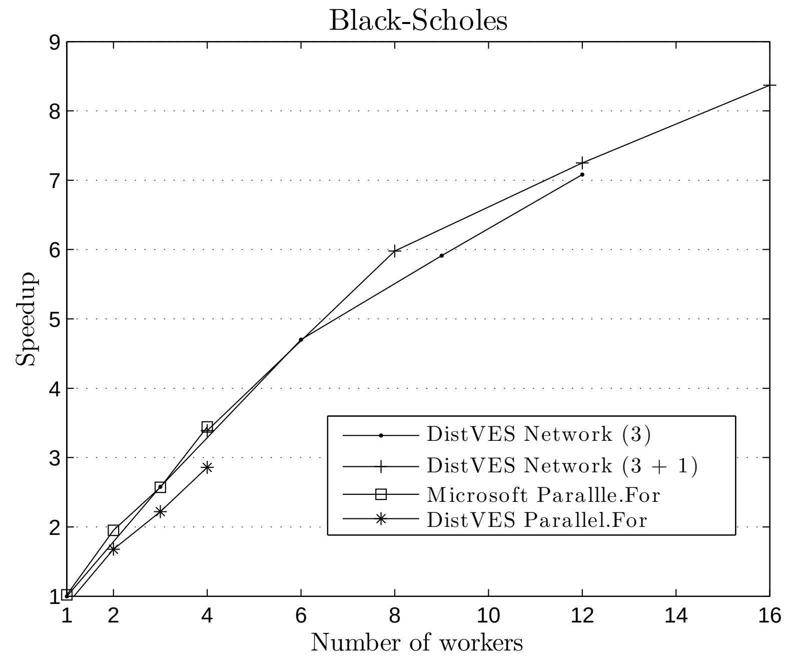


Figure 3.9: Black-Scholes benchmark

embarrassingly parallel, we expect that the DistVES executions will not perform perfectly, because of the large amount of accessed array elements, the synchronization using barriers and finally a low calculation/computation ratio.

As figure 3.10 shows, our predictions holds to a certain degree the TPL library outperforms DistVES. With the many read/write accesses to arrays, the TPL library using shared memory has a clear advantage over the distributed shared memory model used in both the DistVES single machine and DistVES network executions. Please note that even though one could use shared memory for multiple client running on the same machine, DistVES is currently using DSM on all clients also if the client runs on the same machine. This algorithm would properly gain from the implementing the extension were tasks running on the same machine would share a single client (described in the start of this chapter).

3.6 Conclusion

The goal of DistVES is to make it easier for programmers to write parallel code by providing the programmers with a `Parallel.For` construct. By starting a number of DistVES clients on multiple machines, the system distributes the body of the `Parallel.For` method among the clients and execute it in parallel. The programmer does not need to concentrate on shared variables, as they are automatic identified and the system guarantees consistency using a model based on the MESI protocol. As it can be seen from the benchmarks multiple problems exists and like many other tools designed for making it easier to write

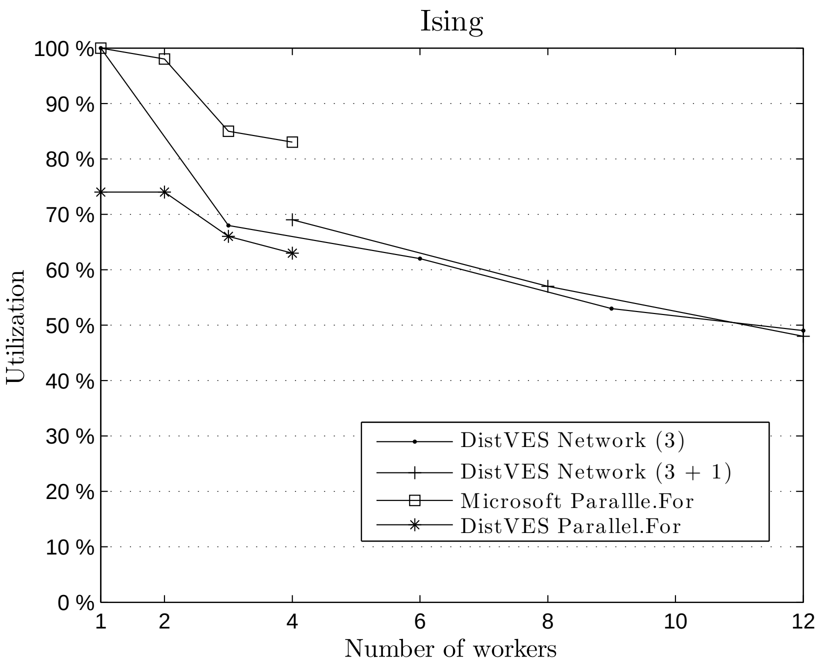


Figure 3.10: Ising benchmark

parallel code, the calculation/communication ratio needs to be of a certain size, in order to be able to hide most of the added overhead related to communicating between machines. In addition, DistVES has problems with the added overhead when accessing array elements, which must be solved in order for the system to be usable. In reality, only one solution in order to minimize the overhead exists and that is to integrate a system like DistVES directly inside the VES. Furthermore, this method would allow DistVES to handle system calls, which would be a huge advantage. Nevertheless, implementing a distributed shared memory system inside the VES is both time consuming and error prone and it would require a lot of expertise about the design and implementation of the VES. Another solution, to minimize overhead is to increase the amount of code analysis done during code generation, so that more information about access patterns could be detected which could lead to the use of a more relaxed consistency model. However, as mentioned earlier this is not easy, as the amount of information available in the CIL code is limited. To increase the amount of information, the code generator could interactively prompt the user for information when doing the code generations. However, none of the methods solves the problem of needing objects to be serialized before being transferred between machines, but if one would make changes to the VES, it would most likely be possible to fix this problem, as the main problem with non-serializable objects is handling references inside an object. Overall, the proposed design and implementation of it, shows that it is possible to make a `Parallel.For` construct that will utilize multiple machines. But, one should also be aware that all benchmark used for testing were simple algorithms and most likely more complicated algorithms would scale poorly, based on some of the trends shown in the implemented benchmarks.

CHAPTER 4

Virtual Execution System in the Cloud

4.1 Introduction

With the introduction of the TPL library for .NET[56], Microsoft provides programmers with a set of tools to simplify the creation of programs that can utilize multicore architectures. Naturally, many programmers already have employed treads for utilizing multicore architectures, but with the introduction of parallel tasks, parallel loops etc. the programmers job is greatly simplified. However, TPL only supports a single machine, and therefore programmers cannot distribute work to a cluster of machines were-of some could be running in a cloud environment. Generally, the amount of tools supporting the .NET programming languages which can help programmers write code that can be executed on multi-core and/or cluster computers are limited and most of them coming from Microsoft. The Microsoft HPC pack[57, 58] allows users to use a mixture of workstations, servers and cloud instances as long as they all use a Windows operating system thereby banning the use of machines running Linux with the support of Mono. Therefore, Microsoft forces programmers who wishes to use such a hardware setup to write their own systems for distributing tasks and data among the machines. Normally, one could use tools like OpenMP,[29, 30] MPI[12] etc., but none of them supports the programming languages of .NET. The proposed system, named CloudVES, simplifies the task of writing programs for programmers understanding the challenges in distributing work and data. The system provides the programmer with a small number of methods, which she can use to distribute work. The goal of the system is to allow programmers to use more of their time on designing the algorithms and considering how data/tasks are associated and less time on the traditional problems of data and task distribution, maintaining consistency, handling fault tolerance etc. The system is based on some of the ideas of Communicating Sequential Processes (CSP)[40] where one designs small simple processes which are connected by channels. However, CloudVES does not adopt the formal proofing possibilities from CSP and therefore the programmer cannot use the tool to guarantee that programs are free of deadlocks and/or race conditions. However, the CSP way of thinking about programs will force the programmers using CloudVES to split algorithms into smaller tasks, which eventually will reduce the risk of the common parallel programming errors.

4.2 Design

This section will describe the design of CloudVES starting with an overview of the components and then a more detailed description of each component. Throughout this chapter the term “task” defines a single work item within a ”job”. The user can submit a job to the CloudVES system and it will then convert the job into a number of individual tasks and then distribute them among the connected workers.

4.2.1 Overview

As mentioned earlier, the goal of CloudVES is to execute code written in one of the .NET languages in parallel on a platform composed of available resources. The thought behind is that a company can send jobs to a system which will execute the jobs in parallel using both available internal resources such as servers, workstations etc., but also external resources from a cloud provider. Normally, systems like CloudVES tries to handle general jobs which can be large executing jobs, small executing jobs, synchronous, loosely synchronous, and asynchronous algorithms. In order to handle all these different types of jobs requires making compromises both in the design and in the amount of information the programmers are forced to provide regarding data access etc. Therefore, CloudVES will only focus on jobs containing many loosely synchronous tasks of any size with the possibility of making dependencies between two or more jobs. The execution platform is meant to consist of a number of machines all capable of executing the intermediate assembly code name (CIL) using either Microsoft .NET or Mono. As cloud instances can be a part of the platform, we must give special attention to security issues. However, in the initial version these concerns are deferred to the “Future work” section along with some other design decisions, which have not yet been implemented. As we expect that each job submitted to the system will produce a large amount of tasks, that would require some time to compute, it must be possible for the user to connect to the system, submit a job, disconnect and then later reconnect to get the result of a given job. Furthermore, multiple users must be able to connect simultaneous and submit/check the status of their jobs. We define the first two components of CloudVES as the client-side component and the server-side component named ”Client” and ”Monitor” respectively. Additional components are the ”Worker” components running on the workstations, servers and cloud instances. Figure 4.1 below shows the overall design of CloudVES, where the components are connected using an interconnect like Gigabit Ethernet, Infiniband etc.

4.2.2 Using CloudVES

The first step is to outline how a programmer can define a job, which the system will execute in parallel. In the TPL library, the programmer can define

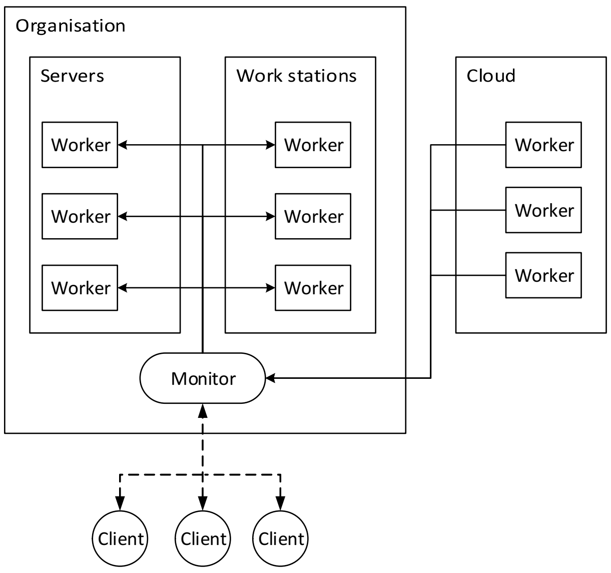


Figure 4.1: An overview of the main components of the tool.

a job using the Task constructors or the Parallel.For loop constructs. TPL defines a task constructor so the programmer can make an array of tasks, which she can start using the blocking Parallel.Invoke method. Whereas the Parallel.For loop executes the body of the loop in parallel. Both methods allows sharing data between tasks. The TPL library does not handle consistency, which is one of the reasons why Microsoft's solution will only run on a single machine. Furthermore, the lack of a consistency model in combination with the absence of explicit defined data access patterns does not force the programmer to consider how data is accessed and more importantly a program can easily be written without considering the risk of race conditions and to some extent dead locks. As seen for MPI and OpenMP, the TPL library bases on the fork-join principal, where the main executing unit distribute (fork) some work to a number of units. Thereafter, the system gathers (join) the results before the main execution continues and perhaps doing additional fork-join operations.

As many other solutions CloudVES use the fork-Join principal, where the fork sends a job containing a single function along with a number of parameters to the monitor. The monitor will then distribute the work among the available workers and return the result to the client when requested. However, unlike other systems, CloudVES requires that the programmer explicitly define which data a task will access, both in terms of the input parameters to the method, but also in terms of accessing shared data. A function send to CloudVES must take zero or more input values and return a single output. Even if the input to a function is a referenced object and the method returns this input as the output, CloudVES sees no correlation between the two. Therefore, the programmer must think of the method as a black box that given a number of inputs will produce one single output. The programmer must pay special attention, if the method modifies one of the input parameters during execution. This would easily result in unintended behaviour, as the changes only reside on the given worker and the system will not propagate changes made to the objects to other workers, unless the input parameters is returned as the output. Therefore, the programmer should only make changes to input parameters if she can ensure that the job does not use the modified input parameters in other tasks during the execution. Even though the programmer can write and compile a method that both read and write shared data, CloudVES will not run this code, as it would require CloudVES to give guaranties about consistency of the shared data.. The reason for this choice is mainly performance related and even though one could construct a consistency model that would handle read/write access, the overhead penalty of maintaining consistency using a system build on top of the .NET framework is too high.[59, 60] At runtime, the monitor will check the code to ensure that any parallel methods does not write to any shared data. By only allowing read access to shared variables and forcing the programmer to use functions that takes a number of inputs and return a single output, ensures that the programmer considers how data is accessed and this will hopefully minimized the risk of the programmer writing code containing race conditions and/or dead locks. The programmer can use the function shown below to execute a method in parallel:

```
Invoke(ID, Method, Parameter/ResultToken, [Selector])
```

The first argument is the ID of the current job and if running multiple associated jobs, the programmer must use the same ID when invoking methods. The second argument is the user-defined method, which the programmer wishes CloudVES to execute and the third argument is the parameters used for calling the method. Finally, the programmer can provide an optional selector used by CloudVES to rearrange parameters before execution. The result of the invocation is a ResultToken defined by CloudVES. The form of defining a parameter is "Type, Start/Ref, [End], [Step]", with End and Step being optional parameters if the type is a numerical type otherwise the system will discard their values. Without an End value, the CloudVES will pass the Start value to all the tasks and the programmer cannot define a Step value. Without a Step value, the value will be set to one by default. Finally, the Start value can be larger than the End value e.g. in combination with a negative step size and a value will wrap if its reach its maximum/minimum values. Moreover, CloudVES allows objects as well as arrays of any dimension as parameters. When the monitor receives a job, it will combine the provided parameters into a list of parameter sets. Given the two parameters (see below), the monitor will produce a list of one hundred (int, double) tuples; namely, the combination of all ten values in the two lists provided:

```
var parameter = new Parallel.Parameters<int, double>(
    Parallel.Parameter.Int.Create(0, 10, 1),
    Parallel.Parameter.Double.Create(0.0, 5.0, 0.5));

((0, 0.0), (0,0.5), (0,1.0), ..., (10,5.0))
```

The system supports three ways of combining parameters, i.e. "element wise", as seen above, "single element", and "zip". Single element means that the monitor will add the provided value to all additional parameters in the list, whereas zip means that the monitor will zip the existing parameters with the values added. The latter is only possible when the array is of the same length as the parameter list. Internally the monitor will define each element in the parameter list as a task, meaning that it will, in the given example, distribute one hundred tasks to the connected workers and each task will produce a single result. This means that the type of the ResultToken is an array where each element has the type of the called method's output parameter. Therefore, the programmer can use the ResultToken as a parameter instead of defining the parameter as shown above. In addition, the programmer can use a Result-Token as a parameter in multiple invocations and the programmer can define dependency between two or more invocations using the token when writing larger algorithms consisting of many methods.

The selector is a function taking an index (integer) as input and based on the input produce a new index. If a given job holds selectors, the monitor execute them on each index of the task queue to produce a new task queue. An example could be that the task queue contained the values a,b,c,d,..,z and the programmer wished to change it so that it would contain (a,b), (b,c) etc. The programmer could then provide two selectors (index => index, index => index +1). When the monitor executes the two selectors on the ith element in the work queue the element would afterwards contain two values namely the values of i and i + 1 elements. The programmer should use selectors a when data must be exchanged between tasks like shown in the shared memory code below, where we wish to run the code a number of times on the same array.

```
for(int i = 0; i < array.Length; i++)
    array[i] = array[1] + (array[i - 1] / array[i + 1]);
```

In CloudVES this could be written using a function taking the whole array and an integer defining "i". The result will be the whole array with a single updated entry namely the i'th entry. However, sharing the whole array when only a small part of it is needed is a waste and a better solution is therefore to make the function take three inputs namely (array[i - 1], array[i], array[i + 1]).

```
Func<int, int, int, int> Worker = (a, b, c) =>
    {
        return b + (a / b);
    };
```

However, as the task queue in the monitor consists of a number of integers (the array split into elements), the programmer must provide three selectors (as seen in figure 4.2) in order to modify the task queue. The output is an array of integers, which the system can yet again modify if the programmer provide a number of selectors.

If a selector returns an out-of-bound index, the monitor inserts the default value of the element, i.e. zero for integers, null for objects etc. However, if a selector returns null the monitor will skip the element. The programmer can use this to reduce the number of elements in a task queue. In order to increase the size of the task queue the programmer must provide a selector returning a tuple containing multiple integers. The monitor will place the i'th element at the indexes defined by the integers in the tuple.

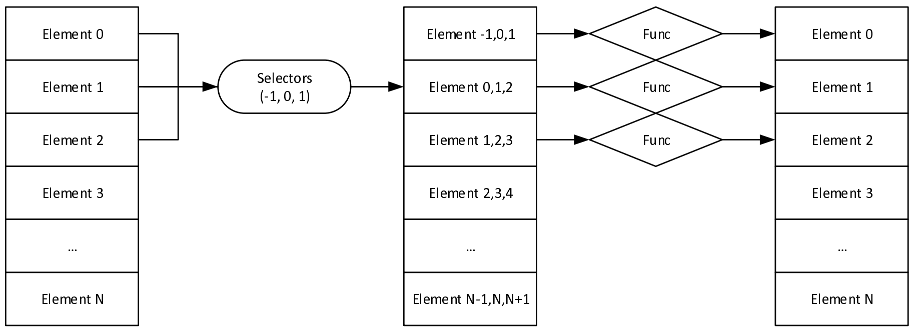


Figure 4.2: Illustrating how three selectors (-1,0,1) can be used to change a work queue.

```
public int SharedField = 100;

public void Run()
{
    // Definer the parameters
    var p1 = new Parallel.Parameters<int, double>(
        Parallel.Parameter.Int.Create(0, 10, 1),
        Parallel.Parameter.Double.Create(0.0, 5.0, 0.5));

    // Define the functions
    Func<int, double, double> Func1 = (v1, v2) =>
    {
        //DoWork
        double temp = v1 * v2;

        return temp;
    };

    Func<double, double> Func2 = (v2) =>
    {
        //DoWork
        double temp = v2 * SharedField;

        return temp;
    };

    // Make an unique ID to identify the job
    var jobID = Guid.NewGuid();

    // Invoke CloudVES and use the token1 as parameter for Func2
    Parallel.ResultToken<double[]> token1 = Client.Invoke(jobID, Func1, p1);
    Parallel.ResultToken<double[]> token2 = Client.Invoke(jobID, Func2, token1);

    // Print whether the result is ready or not
    Console.WriteLine("Result is ready? {0}", token2.IsDone());

    // Block until result is ready
    double[] result = token2.GetResult();
}
```

In order to make it clear how a programmer can use CloudVES, we give a simplified example (see the code above). The first function (Func1) takes an integer and a double and generates a double as result. The second function (Func2) takes a double as input and produces a double as output. In addition, the code illustrates that the programmer can use a token of one method invocation as the parameter for the invocation of another method.

4.3 CloudVES under the hood

Having described how a programmer can use CloudVES it is time to get underneath the hood of the system. We start by assuming that the monitor receives a job from a client, and the rest of this section will describe the steps taken by the monitor and workers to calculate a result, which the monitor will send to the client.

4.3.1 Code modification

After the monitor receives a job, the initial step is to inspect the code of the method, provided by the programmer, for accesses to shared fields and determine that the code obeys the rules of CloudVES. The monitor does this by running through all instructions and insuring that the program does not write to any accessed fields. However, this task is unfortunately not as simple, as one would initial imagine. There exist four CIL instructions to access fields; namely, stfld/stsfld/ldfld/ldsfld which are respectively used to store and load non-static/static fields. One would initially think that the program could just check that no instructions would use the stfld/stsfld. However, if a field contains a referenced object this could be updated by using the load instructions and then calling a method on the object which would update the object in some way. As it would require a recursive traversal of all objects accessed, we decide that a method can only read value-typed fields and arrays of any dimension containing value-typed items. Having identified all accessed fields the monitor can later request the value of each of the fields from the client, then store the values, and forward the values to the workers needing them. With the code analysed for field access, the next step is to figure out how the workers should get the code and how they can execute it. Clearly, the easiest method is just to send the original assembly including the referenced assembly to the worker. However, as we cannot guarantee that the method, which the worker is going to invoke, is static, meaning that the worker can invoke the method without an instantiated target object, the worker must be able to call a constructor and create a target object, which the worker can use to invoke the method. This requires that the monitor modify the original assembly so that it will include an empty constructor, then adding the values of the shared fields before the worker can invoke the method. This sound like a simple task; however, the assembly could potentially contain an empty constructor containing code not related to the user-method. In addition, as CloudVES defines how methods can access shared data objects the result is that a method cannot in any way change the internal state of the target object, we can just take the user-method and insert into, by CloudVES, predefined assembly. The next step is to add the fields accessed by the user-method to the assembly, and afterwards the monitor can create the constructor. The constructor must as a parameter take the values of the fields which we previous got from the client. Thereby allowing the worker to create a target object, which the worker can use to invoke the method defined by the job. The final step is to handle, the parameters used to invoke the method and the result of executing the method. As the parameters come from the monitor and the monitor should be notified when a given tasks has been executed, it would be preferable that the assembly also included code to handle this. Therefore, the monitor add a method to the assembly named Run. This method will contain a While-loop running as long as the workers receive parameters from the monitor. The body of the loop will initially contain two code blocks, i.e. one for requesting a task (parameters) from the monitor and another to send the result of the execution back to the monitor. When the code generator inserts the user-method into the assembly, it also modify the Run method to call the user-method in between the two code blocks. The pseudo code shown above illustrates the Run method. As the user method originally is not included in the executor class and is called without using reflection, each parameter received from the monitor must be casted from an object to the type that match the type of the given parameter. The result of all this is that the worker can create an instance of the executor class, created by the monitor, by invoking the constructor and providing the values of the shared fields. Then the worker can call the Run method of the created class, which will execute the user-defined method until no more tasks are available. Afterwards, the worker can start over by requesting a new job from the monitor.

4.3.2 Task distribution

The next step in the design phase is to consider how CloudVES should distribute tasks to the connected workers. The design of the task distribution method is very essential when performance is of the essence. During the rest of this section, we assume the number of workers, assigned to solve the job, is constant during the whole execution. A task, as described above, is basically a method (a piece of code) and a list of parameter sets and so far the idea is that the workers should only receive the code once, and then continuous receive parameter sets until all sets has been used.

A simple method for centrally distributing the tasks is to let the monitor have a list of all workers together with their current state (working/idle) and then in a round-robin fashion distributed the tasks to the workers with state ”idle”. The opposite solution, even though still centralized, is to let the workers request work from the monitor when they are idle. The advantage, of this approach is that the monitor does not need to maintain a list of workers and their current state, thereby reducing work for the server. To distribute the workload even further CloudVES could implement distributed task scheduling like work stealing.

The basic idea in work stealing is that each worker will have a double-ended queue containing tasks. The worker will process tasks from the top of the queue until it is empty. If a worker has an empty queue, it will try to steal tasks from the bottom of a work queue belonging to one of the other workers. This design will make the scheduling distributed and give automatic load balancing, as the server will do a static distribution by evenly dividing all tasks between the workers. The model do this by inserting tasks in the bottom of the workers work queues and thereby the workers will automatically balance the workload between them. With a design using a double-ended queue minimizes the need for locking the queue as workers process and steal work from each other[61]. The TPL implementation use this approach, but the idea originates from Cilks.

In a system like CloudVES where the focus is on executing many small tasks with low running time, the question is whether work stealing is the right choice. As tasks are short-lived, the risk of doing uneven distribution with a centralized model is small. However, even if, for performance reasons, the scheduler distributes the tasks in chunks of multiple tasks, it could be faster to accept that sometimes workers has nothing to do, instead of trying to steal small task from other clients. Another problem with work stealing is to handle situations where a worker fails or the workstation, server or cloud instance running the worker shuts down or lose network connectivity. All in all, this make implementing work stealing complicated, because the monitor must be able to handle such situations e.g. by having a copy of the distributed tasks which have yet to be computed.

The generation of tasks naturally depends on what input, parameters or a token, the programmer has provided. If the invocation contains parameters, the tasks can easily be generate by combining the parameter values as described above. However, if the invocation contains a result token, the monitor must wait until the job, defined by the token, has completed. In either case, the monitor will use the result of the job defined in the token to generate the new tasks. However, making the workers send the results of all job executed back to the monitor, even if the client does not call GetResult() seems like a waste of resources, as the result of an execution is then most likely to be used as input to another execution and is therefore just an intermediate result. The system should avoid transferring these between the monitor and the workers until the client actually requests a result. However, doing so poses a new problem when creating the token-based tasks, as the monitor distributes the work, it also indirectly knows which worker holds the result of a given task. Therefore, for each distributed task, the monitor needs to record the index of the task in the work queue along with information about the worker having received the task e.g. the workers unique ID and IP-address. The system can now use this information to create tasks and when distributed to the workers, they can use the information to find the location of the data. The worker might have the data itself; otherwise, it must contact the worker holding the result to retrieve the data. This clearly minimizes the amount of data transfers between the monitor and the workers, as we place the data on the workers as long as possible. However, this also poses two new problems; namely, how do we tell the workers that they can delete intermediate data and how does the monitor schedule tasks so in most cases the workers will have the data locally when receiving a task?

Detecting when data is no longer used is impossible in a system like this unless the user explicitly define data as no longer used. The reason for this is that at any time, the programmer could request the result of an execution and the monitor would have no way of guessing this. Likewise, one could argue that if the programmer use a token as an input parameter, the programmer cannot later request the result of the execution. However, this solution has the same flaws as the previous one and as a result; we must require that programmers explicitly define when the monitor/workers can delete intermediate results. This is done by calling the Delete() method on a given token. Still, when the monitor receives a delete request, it can be that the job producing the data or the jobs using the data might not yet have been executed, therefore the monitor should first distributed the delete command to the workers after these jobs has finished execution.

Randomly scheduling tasks, which contains information about where data is located will most of the time, produce fewer data transfers than if the system always transferred all results back to the monitor. However, if algorithms executed using CloudVES shall have a chance of scaling to multiple machines; the workers should do calculation most of the time instead using time on transferring data and communicating with other parties. Doing overlapped execution on workers by executing the current task at the same time as the worker gathers the data required to execute the next task, makes it possible to hide the transfer delay as long as the amount of data transferred is reasonably. However, as the monitor, when scheduling a task, only has information about where the data is located, and the fact that the system allows placing the parameters of a task on different machines, makes this a very difficult task to solve. Furthermore, as the size of the data is unknown to the monitor, it could easily make wrong choices, e.g. by scheduling a task because three of the parameters are located on one machine and the final but much larger parameter is located on a different machine. One solution could be that the workers, when having finished a task report the size of the result back to the monitor, which would help the scheduler make decisions that are more qualified. We could combine this with using a simple round robin scheduling on the monitor and instead letting the workers do the actual scheduling. In this way, if a worker gets a task where most of the data is located on another machine the worker will request the other machine to handle the task. Unfortunately, this easily leads to an unbalanced task distribution, which in turn would reduce the performance. However, we choose this method for the initial version of CloudVES as it is simple to implement.

Transferring tasks from one client to another makes it harder for the monitor to track which worker is executing a given job and thereby finding out where the result for a given task is located. However, the worker, which originally got the task from the monitor, always knows which worker it transferred the task to and therefore, the monitor or worker can always find the result of given task by asking the client who originally got the task from the monitor. This client will be able to redirect the request to the worker having the result. However, as workers with the design mention above already report the size of the result back to the monitor, the monitor will always know which worker have a given result. Both solutions will naturally create a small overhead, as the monitor will have to wait for all results before continuing with distributing the tasks of the next job. Of the two possible solutions, CloudVES currently use the latter.

4.4 Results

The following section will show how the initial implementation of the proposed design will perform using three simple benchmarks. We ran the benchmarks on a small cluster composed of four machines. Each machine was equipped with a quad-core HyperThread Intel i7-860 processor at 2.8 GHz, a minimum of 8 GB of RAM at 1333 MHz and they were connected using a Gigabit network through a single switch. All machines were running Windows 8 with .NET 4.5. All three benchmarks were executed using 1-16 workers (1-4 workers per machine) and performed repeatedly to minimize the uncertainty of the measurements. We distributed the workers in two different ways. Firstly, by filling one machine at the time (round-robin distribution) and secondly, by evenly distributing the workers among the machines. The benchmarks chosen are three algorithms all with no requirements of doing synchronization between tasks and therefore they are relatively simple to parallelize.

4.4.1 Prototein

The first of the benchmarks is the prototein folding, which is a simplified version of the well-known algorithm for folding protein structures.[48] In the simplified version only two types of amino acids (H and P) exists and they can only be positioned in 90-degree angles to one another in a two dimensional plan. The algorithm will, when given a sequence of amino acids try to minimize the number of H amino acids not having another amino acid (H or P) with which is interacts. One can see the folding of a prototein as a large tree where the path from the root node to a given leaf defines one way to fold the prototein. In order to have a reasonable number of tasks, the prototein is partly folded on the client by finding all combinations of placing the first seven amino acids. This will produce around 750 tasks, which the monitor can then distribute among the workers. As there are no dependencies between the two tasks, we expect to the benchmark to scale linearly all the way to sixteen workers allowed by the cluster. The amount of data transferred is very limited and therefore the gradient of the graph should be close to one.

Figure 4.3 illustrates the result from running the prototein benchmark. Both distribution methods give linear scaling; however, when running more than two workers per machine the memory becomes the limiting factor and when running more than two workers per machine only 50 % of the added computational power can be utilized. The figure illustrates that the sequential distribution is close to the optimal line when using one and two workers. The round-robin distribution shows the same tendency. However, the problem can now be scaled to eight workers (two per machine) before the decrease of the gradient is observed. The result is as good as one could expect when memory becomes the limiting factor, and it turns out that the overhead of requesting tasks from the monitor can to a large degree be hidden. Clearly, this does not mean that the system can be scaled to a larger amount of machines each running 1 or 2 workers, but for smaller cluster a 1 % overhead per machine is very acceptable.

4.4.2 Black-Scholes

In the financial world, they use the BlackScholes algorithm for calculation prices on European style options. Like the previous benchmark, the amount of input and output data is very limited and in combination with the algorithm being Monte Carlo base, we expect linear scaling. The result of running the algorithms is seen in figure 4.4 and the result highly resembles the result of running the prototein benchmarks namely linear scaling which is limited by memory and therefore when adding more than two workers per machine the additional cores will only contribute with a utilization of 50 %.

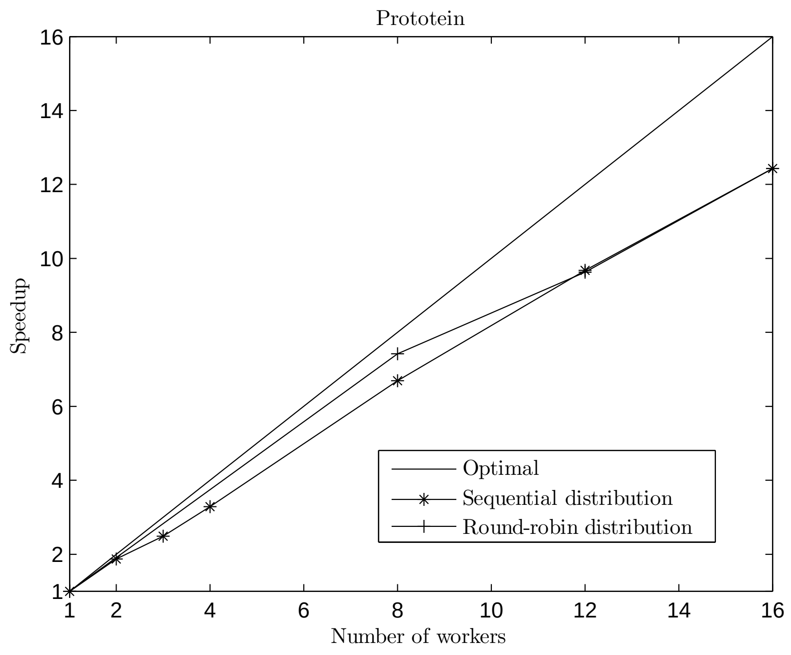


Figure 4.3: Prototein benchmark

4.4.3 k-Nearest-Neighbors

The k-Nearest-Neighbors (kNN) is a well-known problem used in machine learning, and it is used here as the final benchmark for testing the initial version of CloudVES. In this implementation, the algorithm bases on a set of particles of a given dimension, for each particle we wish to find the K nearest particles by calculating the distance between any two particles. As the number of particles required to get a reasonably total execution time is high, letting each particle result in a single task will give a very low calculation/communication ratio. This will eventually lead to low scaling, as the workers will use most of their time requesting tasks from the monitor. Therefore, we divide particles into groups of 1000 particles, which will result in a calculation/communication ratio that is high enough for the solution to scale. Like the previous two benchmark, the result is that the benchmark scale linearly only limited by the memory not being able to get data fast enough to the computational units (see figure fig:CVkNN).

4.5 Future work

This section will include a short description of some of the interesting design idea, which is meant for future implementation.

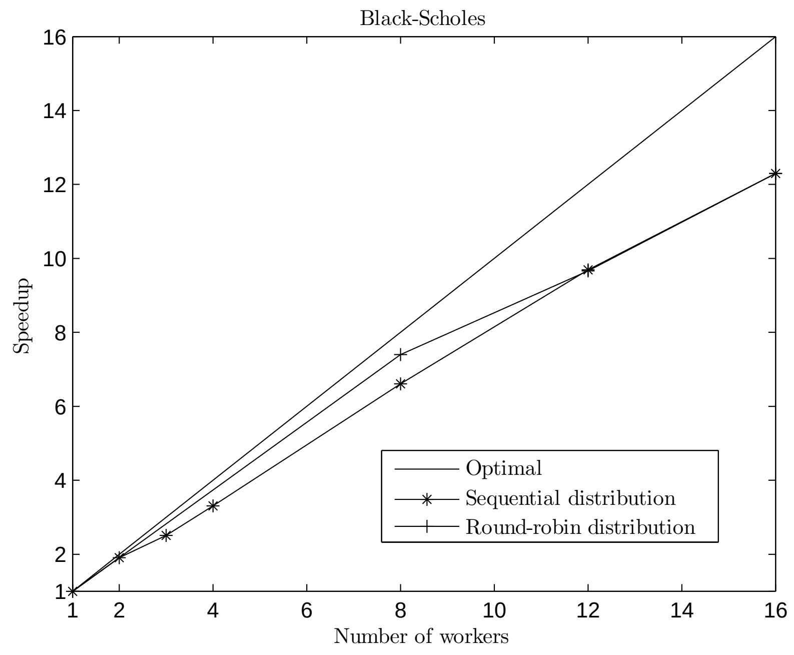


Figure 4.4: Black-Scholes benchmark

4.5.1 Scheduling

As scheduling is one of the most important components in a system like CloudVES it will definitely influence on the total performance of the system. Instead of the workers continuously requesting tasks from the monitor, a better solution is for the monitor to distribute all tasks to a number of workers, and then let the workers schedule work between them using work stealing. An optimal solution is properly doing scheduling where tasks are prioritize based on where the data is located. This means that workers will prioritize executing tasks where they have the data required for executing the task. This can furthermore be combined with the monitor doing intelligent static scheduling based on where data is located e.g. sending all tasks for which most of the data is located on worker "A" to worker "A". This will initially result in an unbalanced task distribution, but if we combine it with the workers being able to steal tasks from each other, this should insure a balanced task distribution. Clearly, we do not need more advanced scheduling if the main use of CloudVES is running simple jobs without dependences like the benchmarks implemented above.

4.5.2 Security

When working with cloud instances, security and privacy of data is one of the major concerns. Therefore, one could imagine that the programmer could mark data as sensitive using attributes, thereby identifying the data, which the programmer does not wish to transfer to cloud instances. Likewise, one must

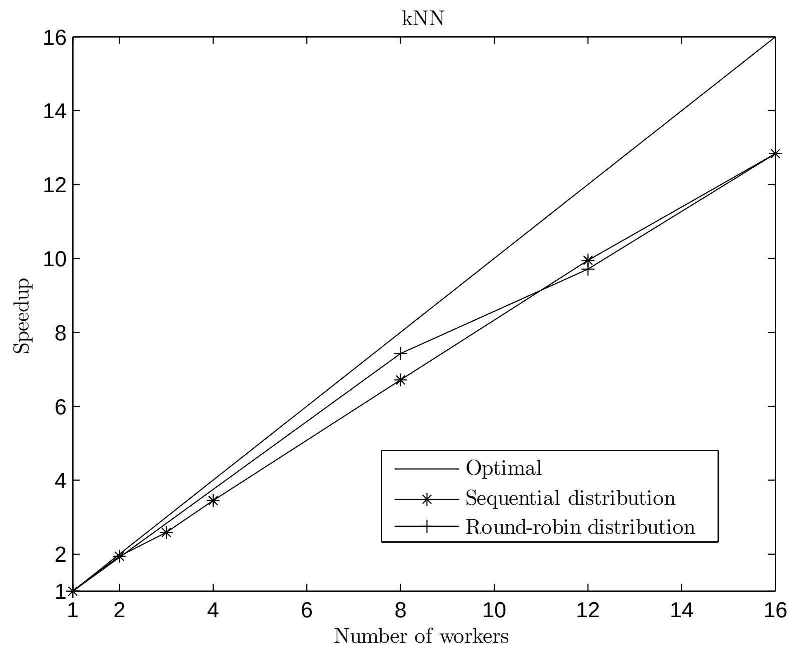


Figure 4.5: k-Nearest-Neighbors benchmark

make considerations about encryption of data and authorization of data and machines to ensure that no one can get access to data or infiltrate the system.

4.5.3 Platform

Cloud providers like Amazon sometimes auctions cloud resources at a reduced price and therefore, situations can occur where it is cheaper to turn off internal resources and mainly use resources from the cloud. In combination with tools for centrally managing workers, this could lead to a reduced cost for companies. Furthermore, future versions of CloudVES should handle situations where machines suddenly are turned off or a person using a workstation running a worker in periods utilizes all the resources of the machine. Both cases, makes scheduling harder and it can become necessary to handle re-scheduling of the same task if a machine dies while or after a calculation is running.

4.5.4 Communication

The current implementation use some of the build-in libraries for network communication and serialization of data. However, faster communication will quickly improve the performance of CloudVES and therefore future implementations should use optimized libraries for handling communication. Likewise, the current implementation assumes that all data can reside in memory on all workers; however, for the system to be truly useful, the system must be able to handle much larger data amount. Other systems, like Hadoop,[62] use a distribute file systems for handling large data amount between machines. With all involved machines having a shared storage, this allows workers to write intermediate results to the shared storage, which naturally changes the requirements to the scheduler, as all machines can access intermediate results from the shared storage.

4.6 Conclusion

The work represented in this chapter will make a good basic for further development of a system for running .NET code in parallel on a mixed platform. Even though we have not implemented some of the ideas, which are critical for real use, the initial testing of the basic functionality of the system clearly shows that it will scale linear using smaller clusters. This does not automatically mean that the system will scale to ten or more machines, but only proves the there is a potential in the idea. Having a central component, namely the monitor, naturally sets a limit of how well the system scales. However, implementing some of the ideas presented in ”future work”, where the central distribution of work is totally distributed by work stealing and using some degree of shared storage, will certainly make it possible for the system to scale further. A much greater and somewhat unsolvable problem is that the implemented benchmarks are memory bound and therefore the gradient of the scaling decreases with a factor two, when running more than two workers per machine. Solving this is very complicated, but as it is very unlikely the workers running on a given machine along with other processes will constantly utilize the memory bus, a possible solution could be to let each worker run two tasks at the same time. When one task is waiting for data, the other could hopefully run unless it also waits for data i.e. the principle of HyperThreading. The presented solution is not perfect and will only work on algorithms that are not completely memory bound, but it provides a good base for future work.

Chapter 5

Conclusion

In the past, the task of writing parallel code was left to highly specialised programmers, which had a good understanding of how large interconnected cluster computers worked and how they should program them. However, as time went by, most workstations started to have the capabilities to run programs in parallel. Therefore, most programmers, including non-computer scientists, must today be able to write parallel code that can utilize the modern architectures with multi-core processors, general-purpose graphic cards etc. This thesis presents a number of tools that can help simplify the process of writing parallel programs especially for the group of non-computer scientists. The target architectures are the CELL Broadband Engine (CELL-BE) along with the more traditionally x86 multi-core architecture. The first tool bases on a previously developed distributed memory system for the CELL-BE, which we extended with a Communicating Sequential Processes (CSP) inspired channel model along with CSP processes. Afterwards, I developed a distributed shared memory model for the Microsoft .NET framework, which allows programmers to run data parallel loops on a small cluster. Finally, I went back to use some of the properties from CSP processes to define a tool for .NET where the programmers indirectly defines data accesses.

The DSM model for the CELL-BE forced programmers to define (very fine-grained) how a program accesses shared data and therefore the benchmarks implemented using the system achieved good performance. Afterwards, when we added a CSP model to the system in order to make it simpler to write programs for the architecture, the limited amount of space available on the specialized computational units made it hard to run the high number of concurrent CSP processes that a CSP program requires. Nevertheless, the system gave good performance showing perfect scaling for embarrassingly parallel programs. However, as the calculation/communication ratio lowers the systems ability to perform naturally decreases as communication becomes the dominating factor. The CELL-BE architecture solved in many ways the problems with the traditionally Von Neumann based architectures especially with the small specialized computational units each having a small memory region with no cache. This meant the programmer was totally in control of managing data, which allowed the programmer to use advanced buffering techniques to overlap execution and data transfers meaning that the memory wall problem was to some degree eliminated. In addition, with being a heterogeneous architecture, this meant that the programming model was very different and therefore very hard to use for many programmers. However, the tool presented in this thesis greatly simplified this, on cost of some performance. However, the question remains if using the CSP extension is actually simpler for programmers than the original implementation and not all scientists will find it easy to break done an algorithm into small parts, which they can convert into CSP processes.

The work done on the .NET resulted in very similar results as the work done on the CELL-BE architecture, both in terms of the tools implemented and the results. However, the DSM model for .NET reduced the amount of information that the tool would force the programmers to provide regarding data access patterns. This made it very easy for the programmers to use the tool, but as consequence, the performance was also limited. The system some not able to scale any of the implemented benchmarks linearly and when using sixteen workers the utilization was around 50 %. The reason for the limited performance is small amount of information making it hard to make release the consistency model and making optimizations. Furthermore, the overhead of accessing shared data was high due to the system being build on top of VES instead of inside the VES. Therefore, the second attempt where the design forces programmers to define how a method accesses data, also resulted in better performance. Systems showed linear scaling on all of the implemented benchmarks when using two workers per machine. Thereafter the scaling decreases a bit giving only a 50 % increase for each additional worker. This decrease is due to the problems being memory bound, meaning that the memory is to slow to provide data fast enough to the computational core. Besides good performance, the programmers was freed of using CSP channels and CSP processes, but still the tool use some of the basic ideas behind these; however, the programming tasks was greatly simplified.

Overall, making tools for helping programmers and scientists write parallel code is challenging, as one must weight all design decisions in terms of wanting performance or usability. Finding a golden path that will yield both high performance and high usability is very unlikely, as more usability means less information provided by the programmer directly influencing on the performance becoming lower. Likewise, higher performance requires more information so that the tool can make good decisions, but this means lowering the usability. In addition, the programmers wish only to write their code once and then be able run it on both the hardware of today, but also on future hardware. Inspired, by the .NET pipeline, one could create a tool that compiles the code to an intermediate language that contained all the information enabling the tool to run the code efficiently. Using an architecture specific just-in-time compiler the tool could compile the intermediate language to machine code. Now, the tool developer only needs to write a new JIT compiler for each new architecture. Furthermore, the tool must support platforms of multiple machines and support ways to combine different architectures. The hardest part, is finding a way where the programmer implicitly defines data access patterns and dependencies without this directly requires more work for the programmer. The best solution for solving this is to make the programmer use a data structure that implicitly defines the parallelism, however, only a subset of data structures, like arrays, have this property. Consequently, such a system only allows programmers to use it for some algorithms namely the ones using arrays.

Appendices

Appendix

A

References

[1] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965.

[2] David A. Patterson and John L. Hennessy. *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition)*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2012.

[3] IBM. Cell broadband engine - programming handbook v. 1.1. Technical report, 2007.

[4] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 177–187, New York, NY, USA, 2009. ACM.

[5] Intel. Automatic parallelization with intel compilers. `http://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers`. Accessed march 2013.

[6] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent programming in ERLANG*, volume 2. Prentice Hall, 1996.

[7] Kenneth Skovhede and Morten N. Larsen. Værktøj til at hjælpe programmører med at håndterer hukommelsesmodellen på cell be architekture. Master's thesis, University of Copenhagen - Specialerapport: 08-02-14, 2008. Written in Danish.

[8] C. J. Conti. Concepts for buffer storage. *IEEE Computer Group News*, 2(8):9–13, 1969.

[9] D. H. Gibson. Considerations in block-oriented systems design. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 75–80, New York, NY, USA, 1967. ACM.

[10] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, Sept.

[11] Wm and Sally A. Mckee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23(1):20–24, 1995.

[12] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. Mpi-2: Extending the message-passing interface. In Luc Bouge, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 128–135. Springer Berlin Heidelberg, 1996.

[13] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. *SIGARCH Comput. Archit. News*, 18(3a):148–159, 1990.

[14] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proc. of the Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'90)*, pages 168–177, 1990.

[15] John B. Carter. Design of the Munin distributed shared memory system. *Journal of Parallel and Distributed Computing*, 29(2):219–227, 1995.

[16] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18:190–205, 1992.

[17] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Pittsburgh, PA (USA), 1991.

[18] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The midway distributed shared memory system. Technical report, Pittsburgh, PA (USA), 1993.

[19] Ahmed K. Elmagarmid. A survey of distributed deadlock detection algorithms. *SIGMOD Rec.*, 15(3):37–45, September 1986.

[20] Christoph Minnameier. Local and global deadlock-detection in component-based systems are np-hard. *Inf. Process. Lett.*, 103(3):105–111, July 2007.

[21] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In Shail Arora and Gary T. Leavens, editors, *OOPSLA*, pages 227–242. ACM, 2009.

[22] Doug Lea. A java fork/join framework. In *Java Grande*, pages 36–43, 2000.

[23] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In Jack W. Davidson, Keith D. Cooper, and A. Michael Berman, editors, *PLDI*, pages 212–223. ACM, 1998.

[24] R. Thurlow. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 5531 (Draft Standard), May 2009.

[25] Microsoft. .net remoting overview. `http://msdn.microsoft.com/en-us/library/kwdt6w2k\%28v=vs.71\%29.aspx`. Accessed march 2013.

[26] Microsoft. Windows communication foundation. `http://msdn.microsoft.com/en-us/library/dd456779.aspx`. Accessed march 2013.

[27] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Incorporated, 2007.

[28] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. 30(8), 1995.

[29] L. Dagum and R Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46, 1998.

[30] Ruud Van Der Pas. An Introduction Into OpenMP. *ACM SIGARCH Computer Architecture News*, 34(5):1–82, 2005.

[31] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[32] Tom White. *Hadoop: The definitive guide*. O'Reilly Media, 2012.

[33] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[34] D. Szafron, J. Schaeffer, and A. Edmonton. An experiment to measure the usability of parallel programming systems. *Concurrency Practice and Experience*, 8(2):147–166, 1996.

[35] L. Hochstein, J. Carver, F. Shull, S. Asgari, and V. Basili. Parallel programmer productivity: A case study of novice parallel programmers. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 35–35. IEEE, 2005.

[36] IBM. A developer's guide to the power architecture. `http://www.ibm.com/developerworks/linux/library/l-powarch/`. Accessed march 2013.

[37] David A Patterson. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, 1985.

[38] J. P. Perez, P. Bellens, R. M. Badia, and J. Labarta. Cellss: making it easier to program the cell broadband engine processor. *IBM J. Res. Dev.*, 51(5):593–604, 2007.

[39] IBM. Accelerated library framework for cell broadband engine programmers guide and api reference. Technical report, 2007.

[40] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[41] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1985.

[42] P. H. Welch, A. W. P. Bakkers (eds, and Nan C. Schaller. Using java for parallel computing - jcsp versus ctj. In *Communicating Process Architectures 2000*, pages 205–226, 2000.

[43] Brian Vinter, John Markus Bjørndalen, and Rune Møllegård Friborg. Pycsp revisited. In *CPA*, pages 263–276, 2009.

[44] Alistair A. Mcewan, Steve Schneider, Wilson Ifill, Peter Welch, and Neil Brown. C++csp2: A many-to-many threading model for multicore architectures, 2007.

[45] Neil CC Brown and Peter H Welch. An introduction to the kent c++ csp library. *Communicating Process Architectures*, 2003:139–156, 2003.

[46] Kenneth Skovhede, Morten N. Larsen, and Brian Vinter. Extending distributed shared memory for the cell broadband engine to a channel model. In Kristján Jónasson, editor, *PARA (1)*, volume 7133 of *Lecture Notes in Computer Science*, pages 108–118. Springer, 2010.

[47] V. Beltran, D. Carrera, J. Torres, and E. Ayguade. Cellmt: A cooperative multithreading library for the cell/b.e. In *High Performance Computing (HiPC), 2009 International Conference on*, pages 245–253, Dec. 2009.

[48] Brian Hayes. Computing science: Prototeins. *American Scientist*, pages 216–221, 1998.

[49] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, 1995.

[50] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, August 1991.

[51] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *In Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

[52] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[53] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report 180-88, Department of Computer Science, Princeton University, sep 1988.

[54] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *SIGARCH Comput. Archit. News*, 12(3):348–354, January 1984.

[55] Intel. Intel 64 and ia-32 architectures software developer's manual. `http://download.intel.com/products/processor/manual/325462.pdf`. Accessed march 2013.

[56] Microsoft. Parallel programming in the .net framework. `http://msdn.microsoft.com/en-us/library/dd460693`. Accessed march 2013.

[57] Microsoft. Microsoft high performance computing for developers. `http://msdn.microsoft.com/en-us/library/ff976568.aspx`. Accessed march 2013.

[58] Microsoft. Microsoft mpi. `http://msdn.microsoft.com/en-us/library/bb524831(v=vs.85).aspx`. Accessed march 2013.

[59] M. Larsen and B. Vinter. A distributed virtual machine for microsoft .net. *Journal of Software Engineering and Applications*, 5(12):1023–1030, 2012.

[60] T. Seidmann. Distributed shared memory using the .net framework. In *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*, pages 457 – 462, may 2003.

[61] Danny Hendler, Yossi Lev, Mark Moir, and Nir Shavit. A dynamic-sized nonblocking work stealing deque. *Distrib. Comput.*, 18(3):189–207, February 2006.

[62] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

Appendix B

Publications

Morten N. Larsen, Kenneth Skovhede, and Brian Vinter. Distributed Shared Memory for the CELL Broadband Engine (DSMCBE). In Leonel Sousa and Yves Robert, editors, ISPDC, pages 121-124. IEEE Computer Society, 2009.

Kenneth Skovhede, **Morten N. Larsen**, and Brian Vinter. Extending Distributed Shared Memory for the CELL Broadband Engine to a Channel Model. In Kristján Jónasson, editor, PARA (1), volume 7133 of Lecture Notes in Computer Science, pages 108-118. Springer, 2010.

Kenneth Skovhede, **Morten N. Larsen**, and Brian Vinter. Programming the CELL-BE using CSP. In Peter H. Welch, Adam T. Sampson, Jan Bækgaard Pedersen, Jon M. Kerridge, Jan F. Broenink, and Frederick R. M. Barnes, editors, CPA, volume 68 of Concurrent Systems Engineering Series, pages 55-70. IOS Press, 2011.

Morten N. Larsen, and Brian Vinter. Distributed Virtual Machine for Microsoft .NET. In Journal of Software Engineering and Applications 2012, 5(12):1023-1030.

Morten N. Larsen, and Brian Vinter. Transparent offloading of parallel components in Microsoft .NET. *Submitted to "Journal of Cloud Computing: Advances, Systems and Applications" on the 8th of March 2013.*

Distributed Shared Memory for the Cell Broadband Engine (DSMCBE)

Morten N. Larsen
Email: morten@momi.dk

Kenneth Skovhede
Email: kenneth@hexad.dk

Brian Vinter
Email: vinter@diku.dk

eScience Centre
University of Copenhagen
Universitetsparken 5
DK-2100 Copenhagen
Denmark

Abstract

The CELL-BE processor provides high performance and has been shown to reach a performance close to the theoretical peak, however, the high performance comes at the price of a quite complex programming model. Central to the complexity of the CELL-BE programming model is the need to move data in and out of non-coherent local storage blocks for each special processor element. In this paper we present a software library, namely the Distributed Shared Memory for the Cell Broadband Engine (DSMCBE). By using techniques known from distributed shared memory DSMCBE allows programmers to program the CELL-BE with relative ease and in addition scale their applications to use multiple CELL-BE processors in a network. Performance experiments show that a quite high performance can be obtained with DSMCBE even in a cluster environment.

1. Introduction

All current computers are fundamentally based on a Von Neumann architecture and all suffer from the 'Von Neumann bottleneck'[1] which boils down to the limited speed of transfer between the memory and the processor. The bottleneck problem was introduced in 1977 and so far no solutions have been provided, only approaches to help hide it.

1.1. STI CELL-BE

The CELL-BE processor attempts to overcome the 'Von Neumann bottleneck' through the use of very high performance specialized processors controlled by a more conventional processor core. Conceptually the CELL-BE is not that different from a small cluster-computer on a chip. Each of the special processors run their own program and has its own memory, although small, and communication between the processors is explicit. The processor cores exchanges information through reading and writing main memory, but with DMA (direct memory access) operations more similar to IO than memory access. From the perspective of the special cores, the main memory is more like a shared storage in a cluster than a conventional main memory. The CELL-BE processor is a heterogeneous multi-core processor consisting of nine cores. The primary core is an IBM 64 bit power processor (PPC64) with two hardware threads. Which is linking the operating system and the 8 powerful working cores, called the synergistic processing elements (SPEs).

1.2. Programming the CELL-BE Processor

As the Cell BE architecture can be viewed as an cluster on a chip with eight nodes, SPEs, with a front-end, the PPE, splitting of an application into smaller tasks is done in the same way as it would be done with traditional clusters. However, due to the limited amount of local storage (LS) on the SPE units available for both code and data, 256 kB, one has to consider the size of each task. This means that an application which would be best parallelized by a bag of task model in a cluster setup, might be best parallelized by a pipelined setup on CELL-BE. If it does not fit into the LS, it might be useful to split it up into two, four or eight pipelined stages depending on the size of the code.

The DSMCBE system helps address this exact problem. Using named memory regions each SPE in a CELL-BE can obtain a copy, possibly exclusive, of a given memory region. A operation that also works as a coordination mechanism. That is, the process may choose to wait until the region is obtained before it continues. The technique is simple since the only difference between mutex programming and DSMCBE is that the locking also migrates the data into the active LS.

The simplicity and flexibility of the DSMCBE model means that it not only works within a single address space for a CELL-BE system but also allows inclusion SPEs on other CELL-BE processors in a cluster, LAN or even WAN environment. Making the DSMCBE system an integrated distributed shared memory system as well as a system for programming CELL-BE processors.

2. DSMCBE

Viewing the CELL-BE processor as a small cluster-computer on a chip, it is straightforward to use a distributed shared memory approach to handle memory. Since the distributed shared memory paradigm is a well researched topic, there is a lot of existing literature[2] and knowledge to apply when building such a system. With DSMCBE we have viewed a cluster of CELL-BE machines as a two level distributed shared memory system. That is, each single CELL-BE processor makes up a single distributed system, but is able to forward requests onto the outer distributed system that connects the cluster of CELL-BE processors. Using this approach, it becomes easier to program a single CELL-BE machine as well as a cluster of machines.

2.1. Data objects

The main objective for DSMCBE is to provide coherent and location transparent memory access. To aid in this, DSMCBE provides a function called `Create` that allows the user to associate a continuous block of memory with a Globally Unique Identifier (GUID). If the blocks are meant to be transferred to the SPE the amount of available memory on the SPE forms an upper limit on the size of the block. Since the SPE units lack memory interrupt handlers, it is not possible to use virtual addressing on the SPE units LS. This means that all data must be allocated as a continuous block on the SPE. This, however, requires that the programmer takes care in deciding the granularity of the objects. A few large objects are desirable because this reduces the overhead pr. byte ratio, but on the other hand memory fragmentation is much more likely to occur if the data objects are large. DSMCBE makes no requirements or assumptions about the data contained in the allocated blocks.

2.2. Object access

Based on research from other DSM systems[3], we have observed that maximum performance can be achieved by using the *Entry Consistency* model[4], which requires that the access to a shared object is marked both for acquisition as well as release. The terms used are functions called `Acquire` and `Release`. Upon a call to `Acquire` the DSM system makes sure that the data requested is updated and available for use. DSMCBE makes sure that only one process may write to a given data object at a time (single writer, multiple reader). When `Release` is called any changes must be propagated to all copies. DSMCBE handles this by issuing invalidate messages. Access to objects is simplified by making all calls blocking including attempts to acquire objects that are non-existent. DSMCBE synchronizes access to data objects by utilizing the *home node* model[5], maintaining a list of available objects, and current copies of each of these objects. Utilization of this model removes the need for broadcasting messages which is critical for obtaining high scalability. Figure 1 shows how DSMCBE is logically structured.

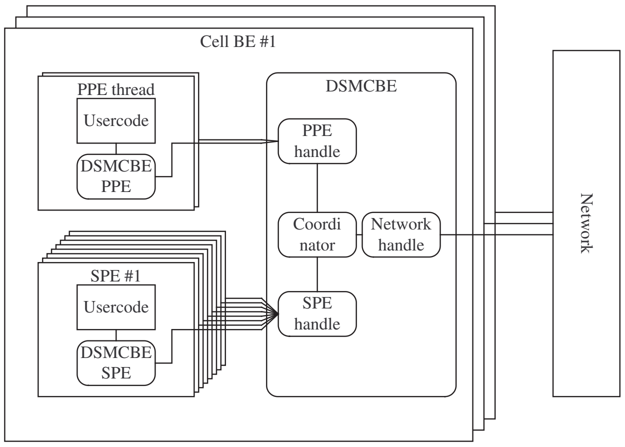


Figure 1. A logical representation of the DSMCBE internal structure.

As the mailbox event system has a large overhead the SPE handlers on the PPE use polling for reading requests from the SPE units. This means that the performance of the SPE units is dependant on how fast the PPE can detect and process the incoming events. On an increase of DSMCBE calls the total system performance depends on the performance of the individual SPE handler threads.

2.3. Thread model

To counter the performance reduction caused by blocking, DSMCBE introduces a thread model for the SPE units. Due to the limited hardware support found on the SPEs the thread model is a user mode, non-preemptive thread model. The thread model is based on the logic found in the UNIX `fork()` function. Using multiple thread stacks it is possible to perform automated context switching instead of blocking on calls. The thread routines are separated and can easily be used in other projects. In the current version of the CELL-BE processor the limitation of the LS means that a large percentage of the memory is occupied by the thread stacks. For applications with high memory requirements on the SPE units the thread model is not desirable. Therefor the user code should use the asynchronous version of the `Acquire` function, which is also supported with DSMCBE.

2.4. Network model

To become a useful system, DSMCBE supports multiple machines connected over a TCP/IP link. Since all objects are accessed through their GUID, all object access is therefor location transparent. Any SPE or PPE thread can request all objects in the system, and DSMCBE will transfer the objects as needed for the operation to succeeded. In the single machine system DSMCBE uses a fixed owner, namely the PPE. Whereas the network enabled DSMCBE system uses a home-based model where the object creator becomes the home node. All objects are kept coherent through the use of invalidate messages, and access is synchronized in a manner that enforces the consistency scheme. The global object table contains the GUID/Machine ID relations which are kept updated using a preselected machine as the home node. The system maintains a cache on each machine as well as a limited cache on each SPE. The multilevel cache system ensures that multiple common access patterns can achieve high performance.

In theory, an internet link could be used to connect several machines using DSMCBE, but since most high-performance computing (HPC) applications are highly sensitive to latency. However this solution is probably not suited for the majority of applications.

3. Performance

This section describes the measurements of the performance of the DSMCBE system when applied to solve standard computational problems. We have chosen a simple problem, ie. prototein folding, and a more complex problem, ie. successive over-relaxation (heat equation).

3.1. Prototein

Protein folding is a well-known problem which is considered embarrassingly parallel. Prototein folding is a simplification in having only two amino acids, which can fold in 90 degree angles. We have constructed a DSMCBE version and a reference implementation using only CELL-BE primitives. The DSMCBE version is executed with single buffer applied manually or with double buffer by the use of threads.

All tests are executed on a cluster of PlayStation 3s connected with a 1Gbit switch. As seen in figure 2 the DSMCBE system performs perfectly, which is to be expected with this kind of embarrassingly parallel problems.

3.2. Successive over-relaxation

Successive over-relaxation (SOR) is a method to approximate the solution of a system of partial differential

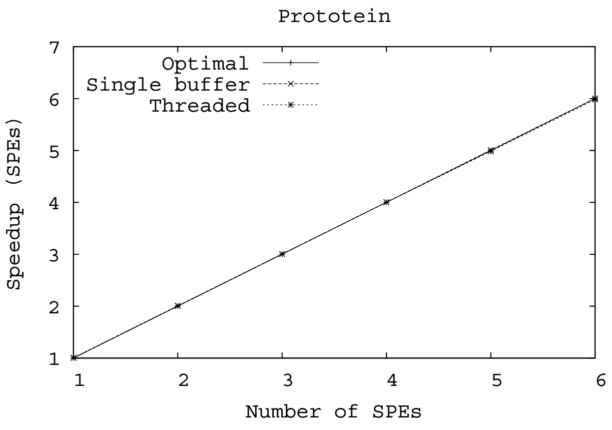


Figure 2. Prototein folding has near optimal performance scaling when using DSMCBE.

equations. Heat equation is a model used to simulated how temperature develops in an object. After applying a given temperature to the edges of the object the heat dispersion is observed in discrete steps. The temperature of any given point on the object is calculated by the average of the temperature of the point plus the temperature of surrounding points. After a fixed number of iterations the simulation stops.

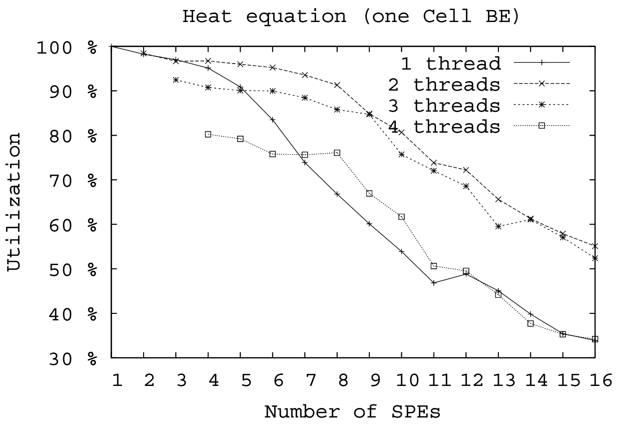


Figure 3. DSMCBE performance on a CELL-BE blade with two CELL-BE cores.

Figure 3 shows the results of running with one to 16 SPEs on a single CELL-BE blade system where the central component in DSMCBE uses one to four hardware threads. Due to the slow handling of events the SPE handling code polls the mailboxes. As the handler thread must service all the SPE units, adding more SPE units increases the amount of time between polling. Since the PPE has multiple threads, performance can be increased by utilizing those threads to service SPE requests. As seen in figure 3, the performance increases significantly when using two hardware threads. It can thus be concluded, that the massive data exchange required for the SOR problem requires a large amount of processing power to perform the administrative tasks. Oppositely performance does not increase any further by adding one additional tread, and by adding yet another tread the performance decreases.

As the test machine is a CELL-BE blade with two CELL-BE cores the performance decreases when using more than eight SPEs.

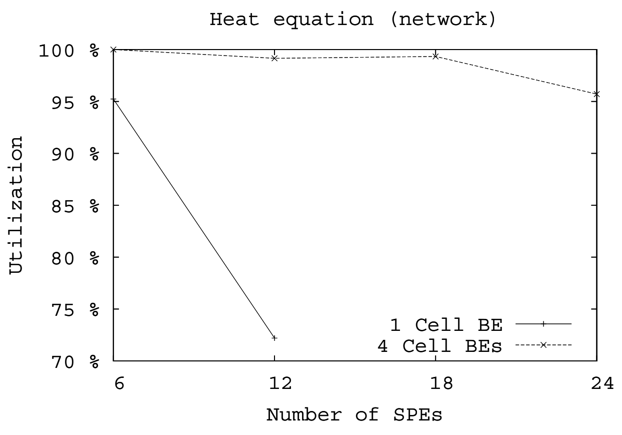


Figure 4. Performance of DSMCBE on small cluster solving a heat equation.

A slight loss of performance is observed, when using two machines as one row of points is shared between the two machines, and therefor they must exchange data. Addition of a another machine result in a further performance decrease as one machine will exchange rows with two other machines. Further increase in the number of machines will cause the performance to decrease till the number of machines and the performance reaches a linearity. Figure 4 shows this linearity of the DSMCBE performance when using two or three machines. However the performance drops a little on addition of the fourth machine.

Figure 4 also contains a graph of the performance when using 12 SPEs on one CELL-BE blade. By comparing the two graphs it is seen that performance of one CELL-BE blade with 12 SPEs is significantly poorer than using 6 SPEs on two CELL-BE blades. This is due to the fact that a single machine is more sensitive to the latency that the DSMCBE is causing, than two machines.

4. Conclusion

DSMCBE is a distributed shared memory system designed especially for the CELL-BE processor with a special focus on the limited memory available to each special processor in a CELL-BE machine. By treating the CELL-BE processor as a cluster-on-a-chip architecture DSMCBE provides programmers with a very simple interface to memory management in the CELL-BE. Since the nature of DSMCBE is treating the CELL-BE as a cluster, DSMCBE extends naturally to several CELL-BE processors including a cheap cluster of Playstation 3 game consoles. Hereby enabling programmers to easily scale their CELL-BE applications beyond a single machine. The performance of applications written using DSMCBE is shown to be quite good, giving linear speedup on optimization problems and high processor utilization on classic scientific computing applications. We believe that the ease-of-use of DSMCBE and the high performance achieved with the system makes it an ideal interface especially for scientific applications on the CELL-BE processor. Work continues on DSMCBE and future versions will seek to allow aggressive migration of memory regions and additional support for profiling DSMCBE applications.

Acknowledgment

We would like to thank the Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation, for the use of Cell Broadband Engine resources that have contributed to this research as well as IBM and Nordisk Film for donation of the Playstation 3 cluster which formed the basis for our development and much of the experiments.

References

[1] J. Backus, “Can programming be liberated from the von neumann style?: a functional style and its algebra of programs,” *Commun. ACM*, vol. 21, no. 8, pp. 613–641, 1978.

[2] A. Judge, P. A. Nixon, V. J. Cahill, B. Tangney, and S. Weber, “Overview of distributed shared memory,” Trinity College Dublin, Tech. Rep., 1998.

[3] P. Guedes and M. Castro, “Distributed Shared Object Memory,” in *Proc. 4th Wshop. on Workstation Operating Systems (WWOS-IV)*. Napa, CA (USA): IEEE Computer Society Press, 1993. [Online]. Available: citeseer.ist.psu.edu/guedes93distributed.html

[4] B. N. Bershad, M. Zerkauskas, and W. Sawdon, “The midway distribted shared memory system,” in *In IEEE COMPCON 93*, 1993.

[5] R. Samanta, L. Iftode, and J. P. Singh, “Home-based svm protocols for smp clusters: design and performance,” in *In Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, 1998, pp. 113–124.

Extending Distributed Shared Memory for the Cell Broadband Engine to a Channel Model

Kenneth Skovhede, Morten N. Larsen, and Brian Vinter

Department of Computer Science, University of Copenhagen,
Universitetsparken 5, DK-2100 Copenhagen, Denmark
{skovhede,momi,vinter}@diku.dk
http://www.diku.dk

Abstract. As the performance gains from traditional processors decline, alternative processor designs are becoming available. One such processor is the CELL-BE processor, which theoretically can deliver a sustained performance close to 205 GFLOPS per processor[10]. Unfortunately, the high performance comes at the price of a quite complex programming model. In this paper we present an easy-to-use, CSP-like, communication method, which enables transfers of shared memory objects. The channel based communication method can significantly reduce the complexity of massively parallel programs. By implementing a few scientific computational cores we show that performance and scalability of the system is acceptable for most problems.

Keywords: CSP, CELL-BE, DSMCBE, channel communication.

1 Introduction

All current computers are fundamentally based on a Von Neumann architecture, and all suffer from the "Von Neumann bottleneck"[1] which boils down to the limited speed of transfers between the memory and the processor. The bottleneck problem was defined by John Backus in 1977, and so far no solutions have been provided to totally eliminate it, only approaches to help hide it.

The CELL-BE processor attempts to overcome the "Von Neumann bottleneck" using highly specialized processors, controlled by a more conventional processor core. Conceptually the CELL-BE is not that different from a small cluster-computer on a chip. Each of the specialized processors run their own program, has their own small memory and communication among the processors are explicit. The processor cores exchange information through reading and writing main memory, but with DMA (direct memory access) operations more similar to IO than memory access. From the perspective of the specialized cores, the main memory is more like a shared storage in a cluster than a conventional main memory. The CELL-BE processor is a heterogeneous multi-core processor consisting of nine cores. The primary core is an IBM 64 bit power processor (PPC64) with two hardware threads, which is linking the operating system and the eight powerful working cores, called the synergistic processing elements

(SPEs). The PPC and the eight SPEs are connected trough a 204.8 GB/s Element Interconnect Bus (EIB)[5]. The computing power of a CELL-BE chip is well investigated[13,9], and a single CELL blade with two CELL-BE processors is reported to yield 460 GFLOPS. This is achieved at a rate of one GFLOPS per second per watt[8].

As the CELL-BE architecture can be viewed as a cluster on a chip with eight nodes, SPEs, and a PPE front-end, splitting an application into smaller tasks are done in the same way as it would be done with traditional clusters. However, due to the limited amount of local storage (LS) on the SPE units available for both code and data, 256 KB, one has to consider the size of each task. This means that an application which would be best parallelized by a bag-of-task model in a cluster setup, might be best parallelized by a pipelined set-up on the CELL-BE. If it does not fit into the LS, it might be useful to split it up into two, four or eight pipelined stages depending on the size of the code.

The DSMCBE[11] system helps address this exact problem. With named memory regions, each SPE can obtain a copy, possibly exclusive, of a given memory region. An operation that also works as a coordination mechanism. That is, the process may choose to wait until the region is obtained before it continues. The technique is simple since the only difference between mutex programming and DSMCBE is that the locking also migrates the data into the active LS.

The simplicity and flexibility of the DSMCBE model means that it not only works within a single address space for a CELL-BE system but also allows inclusion of SPEs on other CELL-BE processors in a cluster, LAN or even WAN environment. Making the DSMCBE system an integrated distributed shared memory system as well as a system for programming CELL-BE processors.

By using DSMCBE, the challenges with the CELL-BE becomes easier to overcome, but it does not lead to flawless concurrent programs. Developers must still be very aware of dead-locks, live-locks and race conditions, which most of the times are very hard to discover at compile time and even at runtime. To overcome these problems C.A.R. Hoare introduced the CSP (Communicating Sequential Processes) data model in 1978[7]. This model is using explicit communication through well defined channels and a concept of processes, each with their own set of private variables. A CSP process can, from a programmers perspective, be seen as a sequential program. Finally it is possible to mathematically prove that a program is free of deadlocks and livelocks[6]. CSP has been implemented on multiple architectures and in many programming languages e.g., occam, C++CSP[12], JCSP[14] and PyCSP[3], but not yet on the CELL-BE. This paper describes how to utilize the functionality of DSMCBE to make CSP communication channels for the CELL-BE.

2 Work

One essential property of CSP, is the ability to move data to the processes that require it. As we have previously made a DSM system for the CELL-BE, we have used that as a base for the CSP channel implementation.

2.1 DSMCBE

DSMCBE is an usermode library that offers a simple API for working with shared memory regions. The user explicitly calls the three functions `create`, `acquire` and `release`. Using these three functions, DSMCBE can implement release consistency, which makes it possible for programs to share memory regions reliably. The underlying transport of data is completely hidden from the usercode, giving the programmer the illusion that memory is shared. Performance and scalability are shown to be quite good for most computational problems[11].

The DSMCBE library consist of several elements. The most central element is a single processing thread called the request-coordinator. Most of the other elements of DSMCBE can communicate with the request-coordinator by supplying a target for the answer. Using this single thread approach makes it simple to execute atomic operations and reduce the total number of locks required. The request-coordinator cannot determine if a participant is a PPC, SPE or the network[1]. The main drawback for this simple single thread design is that the thread may become a bottleneck.

Figure 1 shows the logical parts of the DSMCBE library.

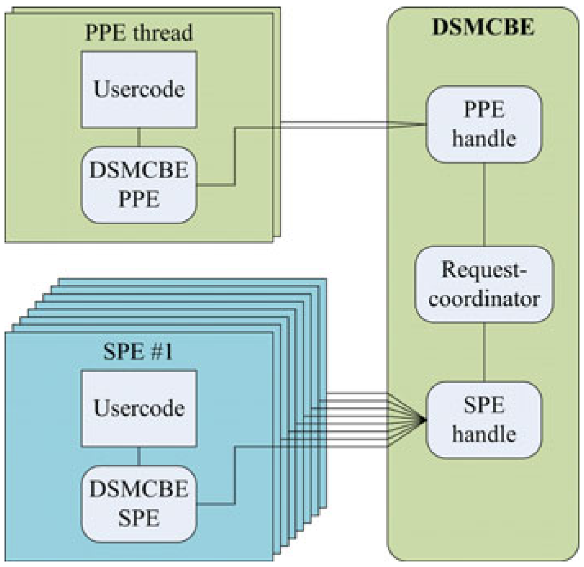


Fig. 1. DSMCBE internal structure

2.2 Channels

Using a DSM system as the base for a CSP channel implementation makes the data transfer operations simple, as they can use the DSM system directly. As CSP channels are blocking, the DSM system must support blocking in some way. Even though the DSMCBE system supports blocking multiple writers, it

[1] The network component is not shown in figure 1.

does not support resizing or deleting of shared regions. Therefore we have implemented a blocking create option to DSMCBE, so that multiple create calls are blocked until the shared region can be re-created. To ensure that the create calls can be executed we have also implemented a delete operation that works as an `acquire` call in write mode, but marks the object as deleted. This also enables the processes to re-create the object with a different size. Using the blocking `create` and `acquire-delete` calls, it is possible to implement a simple CSP channel. Since it is possible to create a shared region without a pending `acquire`, it is essentially a channel with a buffer size of one. This means that it is not possible for the programmer to define a unbuffered channel.

Figure 2 shows the DSMCBE library extended with the channel wrapper.

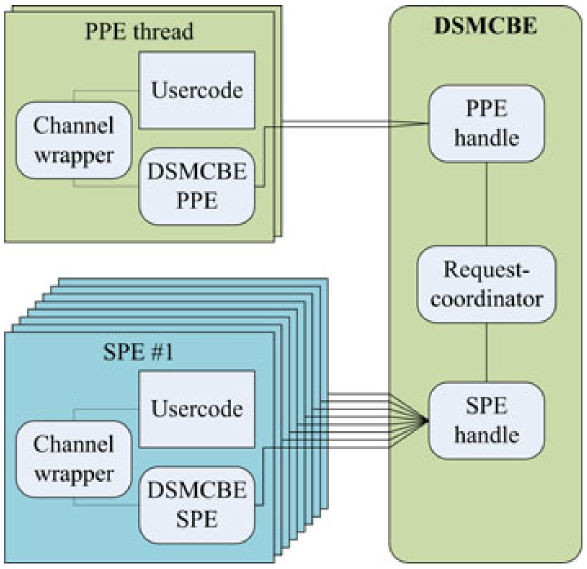


Fig. 2. DSMCBE with channel wrapper internal structure

2.3 Implementation

To support a channel based communication, we have modified the `create` call for the DSMCBE system to accept a blocking strategy for `create` call to items that already exist. To ensure that the `get` function call is atomic, we have implemented a `delete` function that removes and returns an existing item. As the channel model means that the `get` call is expected to block, the `delete` call also blocks until an item is created.

Rather than implement specific control structures for the `get` and `put` requests, they are merely implemented as simple wrapper functions that call the relevant version of the `delete` or `create` request respectively. As the `create` and `delete` calls are ordinary DSMCBE calls, they are actually completely asynchronous, simply forwarding a control structure. To adhere to the expected behavior of channels, the `get` and `put` operations will immediately block until a response is received, thus appearing to be normal synchronous calls.87

As shown in figure 1 all requests from a SPU is forwarded to a *SPU handler* on the PPE, which is done using mailboxes. The *SPU handler* then forwards this request to the *request-coordinator*, which is in charge of maintaining the state and request queue for each item. When a request can be serviced, the *request-coordinator* will send a response back to the *SPU handler*, which will initiate DMA requests and ultimately respond to the SPU.

Internally in the DSMCBE model, each request is recorded in a simple FIFO queue, making most operations run in constant time. This ensures that the number of pending `put` requests do not affect the execution time.

3 Results

Using the simple channel communication system, we have implemented a few experiments, that show how well the system performs and scales. For each application there are some peculiarities that stem from the special hardware construction that is found in the CELL-BE. To establish a realistic scenario for the intended library usage, we have chosen one CSP experiment and two representative scientific computational cores. The first experiment is the CommsTime which is widely used to measure the overhead in CSP implementations. The second experiment (Prototein folding) is used to measure the implementations capability of scaling. The Prototein folding application is a basic bag-of-tasks solution that can be classified as an embarrassingly parallel application. The final experiment is the Heat Equation application which has an entirely different type of communication pattern and is an instance of the successive over-relaxation solutions. This experiment is used to see how the implementation reacts on heavy communication. The code for all experiments presented, as well as the code for the DSMCBE and CSP model can be found on the DSMCBE website `http://code.google.com/p/dsmcbe`

3.1 CommsTime

To measure the overhead of using the implemented CSP channels, we have chosen CommsTime which is a well-known method to benchmark CSP systems. CommsTime is used to compute the cost of a single channel communication operation, and thereby it is possible to compare this implementation against other CSP implementations.

As the CELL-BE processor has the eight available SPEs, we have executed CommsTime with two to eight SPEs participating. The first SPE will run the delta process that forwards the message and outputs the clock signal. The second SPE will run the prefix process and inject the value to send around. Any additional SPEs will run a delta process with one output, and just forward the message. Each additional SPE will add a communication channel. The PPE reads the clock signal from a channel and measures the time between each clock signal. Figure 3 shows the conceptual setup with four participating SPEs.

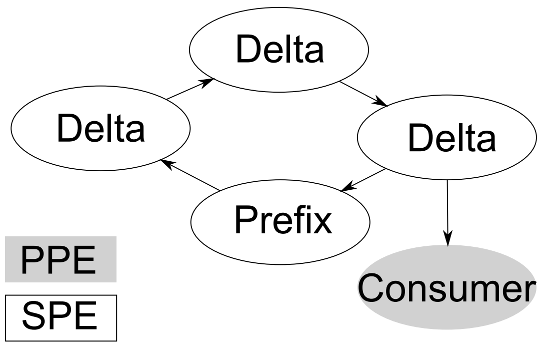


Fig. 3. CommsTime conceptual diagram with four SPEs

Figure 4 shows how the CommsTime application performs and shows that the CommsTime improve after two SPEs. This happens because the first SPE outputs the delta signal, which means that it has two channels, where any additional process has only one outbound channel. After this initial stage, the times are fairly constant, with a slight increase in time. This happens because the actual computation time on each SPE is very limited, and each SPE thus awaits a PPU service most of the time. As there is only one PPU thread to service the SPEs, the time between the service calls increase with the number of SPEs. To remove bottleneck in the PPE from the system requires a change in the DSMCBE. To help remove this bottleneck, one could try to use multiple threads in the DSMCBE *SPE handler* module. This could improved the performance because the SPE handler is in a spin-loop the most of the time. However, multiple threads also requires more synchronization which do not improved performance. A better solution would be to remove the spin-lock, but this is not possible due to the hardware structure of the communication between the PPE and SPEs.

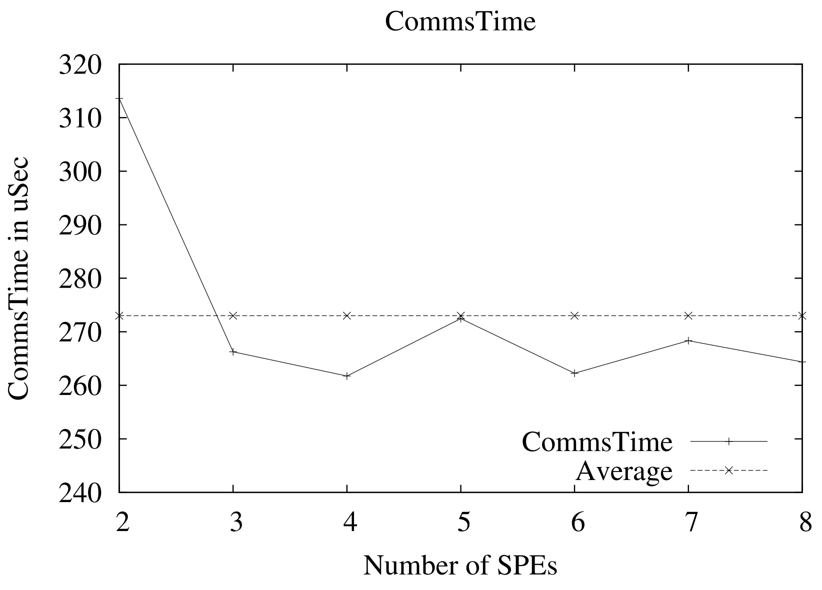


Fig. 4. CommsTime performance using one to eight SPEs

A peculiar effect of the test is that the CommsTime problem runs a little slower when then number of participating processes are odd. We expect that this is due to a transfer aliasing effect, and is subject of further investigation.

Table 1 shows the CommsTime measure for several systems, and reveals that the communication time is quite high compared to other CSP implementations. This large communication time is mainly caused by the physically separated memory blocks that require memory transfers. The number of messages passed internally in the DSM system is also larger than the required minimum. This is because the DSMCBE system supports multiple readers, which is not used in the channel communication scenario.

Table 1. Performance of one CSP channel communication using several CSP frameworks [4]

CSP Framework	Time per iterations (microSec)
CSP for CELL-BE (avg.)	273
OCCAM (KRoC 1.3.3)	1.3
C++CSP	5.0
JCSP (JDK 1.4)	230

One important difference between the CELL-BE CSP channel implementation and the other implementations, is that for the other implementations, the measured overhead is execution time on the processor that runs both user code and CSP library code. In the CELL-BE implementation this is not the case, as the CSP user code runs on the SPE units and the library code runs (mostly) on the PPU. This means that even though the overhead is large compared to the other implementations, the overhead runs in parallel with the user code, and will thus be hidden in many real-life applications.

3.2 Prototein

A prototein is a model of a protein that only contains two amino acids and only folds in 90 degree multiples. Folding a large prototein is a computationally intensive task, but is embarrassingly parallel since the subtasks have no interdependencies. We have implemented a single channel to dispense the subtasks, making it a bag-of-tasks type of implementation, with a single writer and multiple readers. The PPU is responsible for the initial problem division, and writes partially folded prototeins into the channel. Each participating SPU reads the partially folded prototeins from the same channel and completes the fold. Each SPU reports the best possible fold back to the PPE, which then picks the overall best fold. Figure 5 shows a conceptual setup of the communication pattern in the prototein folding.

As seen in figure 6, the prototein problem scales close to perfectly. The channel based implementation is slightly faster than the DSM implementation, because the bag(-of-tasks) is a shared object in the DSM model, where this is handled by

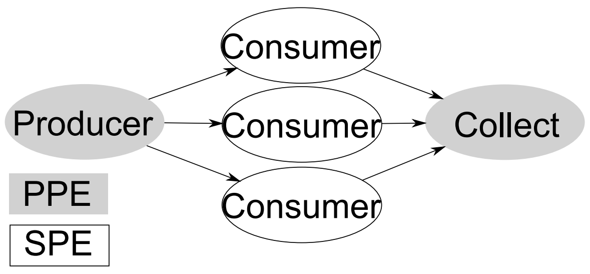


Fig. 5. Prototein conceptual diagram with three SPEs

blocking the requests in the channel version. This also means that the channel based model scales marginally better. The PPU service problem is not as present as in the CommsTime test because the SPEs actually do some computation and do not require the PPU service as often.

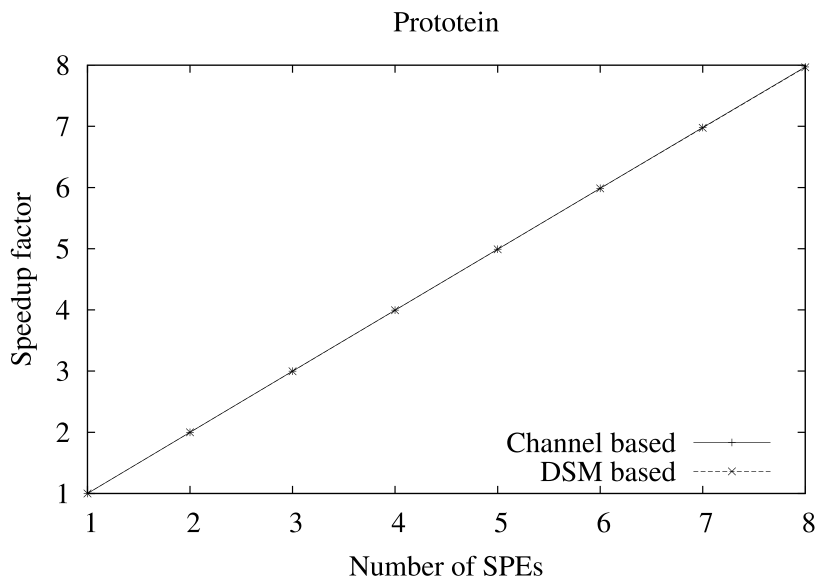


Fig. 6. Prototein folding has near optimal performance scaling

3.3 Successive over-Relaxation

In the final experiment we have implemented an example of successive over-relaxation (SOR) where a large block is initialized with a temperature of zero in the middle and -273.15 on the sides. The temperatures movement in the block is then simulated by successive over-relaxation in discrete time steps. After 1000 iterations, the simulation is stopped.

Each SPU is responsible for handling a fragment of the total simulated area, which is managed with double buffered transfers through the DSMCBE system. The shared boundaries between the fragments are coordinated through a channel for each SPU pair. The first and the last SPU has a single channel, where all others have two channels, one for the upper shared boundary, and one for the lower. Each half iteration consists of applying a SOR for one half of the points, followed by an exchange of boundaries. A full iteration is performed with repeating the half-iteration twice, each with a different half of the points. Figure 7 shows the conceptual communication pattern for the SOR sample application, and illustrates the lock-step setup.

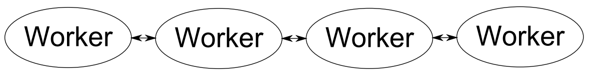


Fig. 7. HeatEquation conceptual diagram visualizing worker dependence

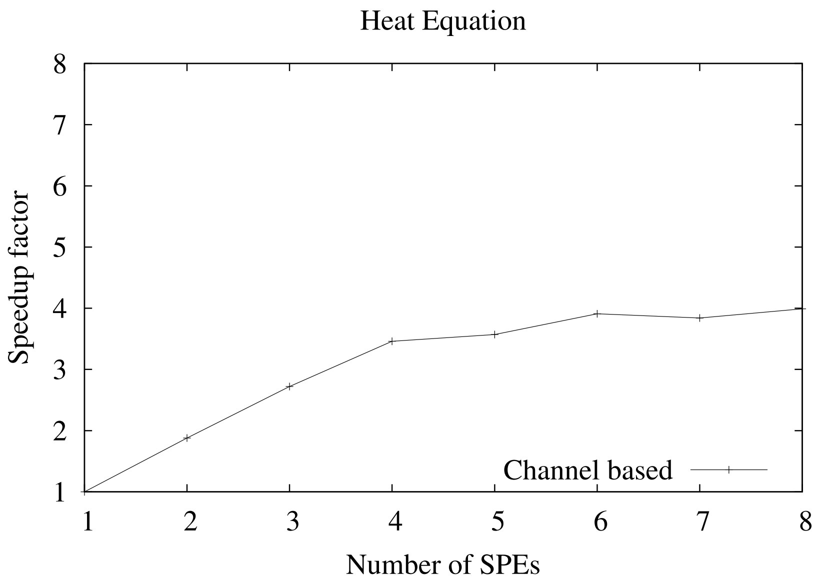


Fig. 8. HeatEquation suffers from contention on PPU resources

Figure 8 shows how the CSP model scales with one to eight SPE units, utilizing two PPU threads. As can be seen the problem does not scale well beyond four SPEs. The hard time constraints inherent in the problem, means that before each round can begin, all the processes must synchronize. This results in the slowest process preventing faster processes from completing.

Since the objects that we simulate are so large that they cannot fully fit in memory on the SPE units, we use the DSMCBE system to load memory regions onto the SPE units. This loading uses the same system as the channel communication, which naturally influences the measured times.

The overall problem with scalability in this example is the amount of service required from the PPU. Even though the SPE units handle the actual computation, they rely on the PPU for transferring both shared and private blocks. When the number of participating SPEs increase, so does the number of requests to the PPE. As a single slow SPE blocks all other SPEs, any delay in handling the requests propagate, resulting in the scalability problem seen in figure 8.

4 Conclusion

The absence of race-conditions in a channel based communication pattern, significantly reduces the program complexity. The reduced complexity comes at the expense of a drop in performance. Even though the overhead in the channel model is fairly high, the scalability is quite good. The scalability combined with the massive processing power offered by the SPE units, makes the library attractive for a number of tasks such as protein folding.

Clearly problems that are computationally intensive but have low memory requirements scale very well on the CELL-BE. When the problems have large memory requirements, the data transfers become the bottleneck. As DSMCBE is capable of performing asynchronous transfers, the transfer times can be hidden by performing overlapped execution. Hopefully overlapped execution can be exploited when buffered channels are implemented. There exists a boundary for the smallest number of computations done on a single byte, before the bytes transfer time can be hidden. What this boundary is has not been examined, but is excepted to be around 20-30 floating point operations per byte.

All work on DSMCBE and the CSP implementation, including the experiments, is released as open source under the LGPL license, and is available from the website: `http://code.google.com/p/dsmcbe`

5 Future Work

The underlying DSM system is optimized for multiple readers and multiple writers, which means that the object is recorded and managed in multiple places. Since channel objects can only exist in one place, there is a large potential for optimization. Once the basic channel system is in place, it would be desirable to implement a full CSP model with alternating channels. To ease the use of the channels it would be desirable to use a common standard for channels e.g., the one used in PyCSP, JCSP.

With CSP channels for the CELL-BE the next step is to make CSP processes. With the DSMCBE functionality to transfer data among the CELL-BEs, it would make good sense to use DSMCBE to transfer the CSP processes (user code). Once the code has arrived at the designated execution unit it can be executed. Some work relating to relocation of processes has already been done, but making a “real” thread library for the SPEs is not a trivial task. The fact that the SPEs do not have more then 256KB of LS, means it is possible that threading is not a feasible solution when we want many (100+) threads. Some work have however been done in this field[2], but at the time being it only supports a small number of threads.

Acknowledgments. The authors acknowledge the Danish National Advanced Technology Foundation (grant number 09-067060) and the innovation consortium (grant number 09-052139) for supporting this research project. Furthermore the authors acknowledged Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation, for the use of Cell Broadband Engine resources that have contributed to this research.

References

1. Backus, J.: Can programming be liberated from the von Neumann style?: A functional style and its algebra of programs. Commun. ACM 21(8), 613–641 (1978)
2. Beltran, V., Carrera, D., Torres, J., Ayguade, E.: CellMT: A Cooperative Multithreading Library for the Cell/B.E. In: The Proceedings of the 16th Annual IEEE International Conference on High Performance Computing, HiPC 2009 (December 2009)
3. Bjørndalen, J.M., Vinter, B., Anshus, O.: PyCSP - Communicating Sequential Processes for Python
4. Brown, N., Welch, P.: An Introduction to the Kent C++CSP Library. Slides
5. Chen, T.: Cell Broadband Engine Architecture and its first implementation - A Performance View (2005), `http://www.ibm.com/developerworks/power/library/pa-cellperf/` (accessed July 26, 2010)
6. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM 21(8), 666–677 (1978)
7. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
8. IBM: IBM doubles down on cell blade (2007), `http://www-03.ibm.com/press/us/en/pressrelease/22258.wss` (accessed July 26, 2010)
9. Jowkar, M.: Exploring the Potential of the Cell Processor for High Performance Computing (2007), `http://www.diku.dk/~rehr/cell/docs/mohammad_jowkar_thesis.pdf` (accessed July 26, 2010)
10. Langou, J., Langou, J., Luszczek, P., Kurzak, J., Buttari, A., Dongarra, J.: Exploiting the Performance of 32 bit Floating Point Arithmetic in Obtaining 64 bit Accuracy (2006), `http://www.netlib.org/lapack/lawnspdf/lawn175.pdf` (accessed March 29, 2010)
11. Larsen, M.N., Skovhede, K., Vinter, B.: Distributed Shared Memory for the Cell Broadband Engine (DSMCBE). In: International Symposium on Parallel and Distributed Computing, pp. 121–124 (2009)
12. Mcewan, A.A., Schneider, S., Ifill, W., Welch, P., Brown, N.: C++CSP2: A Many-to-Many Threading Model for Multicore Architectures (2007)
13. Rehr, M.: Application Porting and Tuning on the Cell-BE Processor (2008), `http://dk.migrid.org/public/doc/published_papers/nqueens.pdf` (accessed July 26, 2010)
14. Schaller, N.C., Hilderink, G.H., Welch, P.H.: Using Java for Parallel Computing - JCSP versus CTJ. In: Communicating Process Architectures 2000, pp. 205–226 (2000)

Programming the CELL-BE using CSP

Kenneth SKOVHEDE [a,1] Morten N. LARSEN [a] and Brian VINTER [a],
[a] *eScience Center, Niels Bohr Institute, University of Copenhagen*

Abstract. The current trend in processor design seems to focus on using multiple cores, similar to a cluster-on-a-chip model. These processors are generally fast and power efficient, but due to their highly parallel nature, they are notoriously difficult to program for most scientists. One such processor is the CELL broadband engine (CELL-BE) which is known for its high performance, but also for a complex programming model which makes it difficult to exploit the architecture to its full potential. To address this difficulty, this paper proposes to change the programming model to use the principles of CSP design, thus making it simpler to program the CELL-BE and avoid livelocks, deadlocks and race conditions. The CSP model described here comprises a thread library for the synergistic processing elements (SPEs) and a simple channel based communication interface. To examine the scalability of the implementation, experiments are performed with both scientific computational cores and synthetic workloads. The implemented CSP model has a simple API and is shown to scale well for problems with significant computational requirements.

Keywords. CELL-BE, CSP, Programming

Introduction

The CELL-BE processor is an innovative architecture that attempts to tackle the problems, that prevent processors from achieving higher performance [1,2,3]. The limitations in traditional processors are primarily problems relating to heat, clock frequency and memory speed. Instead of using the traditional chip design, the CELL-BE consists of multiple units, effectively making it a cluster-on-a-chip processor with high interconnect speed. The CELL-BE processor consists of a single PowerPC (PPC) based processor connected to eight SPEs[1] through a 204.8 GB/s EIB[2] [4]. The computing power of a CELL-BE chip is well investigated [5,6], and a single CELL blade with two CELL-BE processors can yield as much as 460 GFLOPS [7] at one GFLOPS per Watt [7].

Unfortunately, the computing power comes at the price of a very complex programming model. As there is no cache coherent shared memory in the CELL-BE, the processes must explicitly transfer data between the units using a DMA model which resembles a form of memory mapped IO [8,4]. Furthermore to fully utilize the CELL-BE, the application must use task-, memory-, data- and instruction-level (SIMD[3]) parallelization [5]. A number of papers discuss various computational problems on the CELL-BE, illustrating that achieving good performance is possible, but the process is complex [5,9,10]. In this paper we focus on the communication patterns and disregard instruction-level and data parallelization methods because they depend on application specific computations and cannot be easily generalized.

C.A.R. Hoare introduced the CSP model in 1978, along with the concept of explicit communication through well-defined channels. Using only channel based communication, each

[1]Corresponding Author: E-mail: `skovhede@nbi.ku.dk`.
[1]Synergistic Processing Elements, a RISC based processor.
[2]Element Interconnect Bus.
[3]Single Instruction Multiple Data.

participating process becomes a sequential program [11,12]. It is possible to prove that a CSP based program is free from deadlocks and livelocks [11] using CSP algebra. Furthermore, CSP based programs are easy to understand, because the processes consist of sequential code and channels which handle communication between the processes. This normally means that the individual processes have very little code, but the total number of processes are very high.

This work uses the CSP design rules and not the CSP algebra itself. By using a CSP like interface, we can hide the underlying complexity from the programmer giving the illusion that all transfers are simply channel communications. We believe that this abstraction greatly simplifies the otherwise complex CELL-BE programming model. By adhering to the CSP model, the implementation automatically obtains properties from CSP, such as being free of race-conditions and having detectable deadlocks. Since the library does not use the CSP algebra, the programmer does not have to learn a new language but can still achieve many of the CSP benefits.

1. Related Work

A large number of programming models for the CELL-BE are available [13,14,15,16] illustrating the need for a simpler interface to the complex machine. Most general purpose libraries cannot be directly used on the CELL-BE, because the SPEs use a different instruction set than the PPC. Furthermore, the limited amount of memory available on the SPEs makes it difficult to load a general purpose library onto them.

1.1. Programming Libraries for the CELL-BE

The ALF [13] system allows the programmer to build a set of dependent tasks which are then scheduled and distributed automatically according to their dependencies. The OpenMP [14] and CellSs [15] systems provide automatic parallelization in otherwise sequential code through the use of code annotation.

As previously published [16], the Distributed Shared Memory for the CELL-BE (DSM-CBE), is a distributed shared memory system that gives the programmer the "illusion" that the memory in a cluster of CELL-BE machines is shared. The channel based communication system described in this paper uses the communication system from DSMCBE, but does not use any DSM functionality. It is possible to use both communication models at the same time, however this is outside the scope of this paper.

The CellCSP [17] library shares the goals of the channel based system described in this paper but by scheduling independent processes with a focus on processes, rather than communication.

1.2. CSP Implementations

The Transterpreter [18] is a virtual machine that can run occam-π programs. By modifying the Transterpreter to run on the SPEs [19], it becomes possible to execute occam-π on the CELL-BE processor and also utilize the SPEs. The Transterpreter implementation that runs on the CELL-BE [19] has been extended to allow programs running in the virtual machine to access some of the SPE hardware. A similar project, trancell [20], allows a subset of occam-π to run on the SPU, by translating Extended Transputer Code to SPU binary code.

Using occam-π requires that the programmer learns and understands the occam-π programming language and model, and also requires that the programs are re-written in occam-π. The Transterpreter fro CELL-BE has an extension that allows callbacks to native code [19], which can mitigate this issue to some extent.

A number of other CSP implementations are available, such as C++CSP [21], JCSP [22] and PyCSP [23]. Although these may work on the CELL-BE processor they can currently only utilize the PPC and not the high performing SPEs. We have used the simplified channel interface in the newest version of PyCSP [24] as a basis for developing the channel communication interface. Since DSMCBE [16] is written in C, we have produced a flattened and non-object oriented interface.

2. Implementation

This section gives a short introduction to DSMCBE and describes some design and implementation details of the CSP library. For a more detailed description and evaluation of the DSMCBE system see previous work [16].

2.1. Distributed Shared Memory for the CELL-BE (DSMCBE)

As mentioned in the introduction, the basis for the implementation is the DSMCBE system. The main purpose of DSMCBE is to provide the user with a simple API that establishes a distributed shared memory system on the CELL-BE architecture. Apart from its main purpose, the underlying framework can also be adjusted to serve as a more generic platform for communication between the Power PC element (PPE) and the Synergistic Processing Elements (SPEs). Figure 1 shows the DSMCBE model along with the components involved. The DSMCBE system consists of four elements which we describe below:

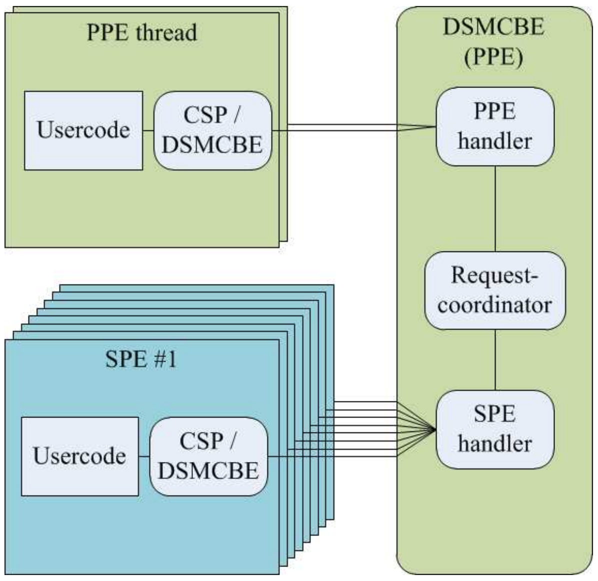


Figure 1. DSMCBE Internal Structure.

The DSMCBE PPE/SPE modules contains the DSMCBE functions which the programmer will call from the user code. To manipulate objects in the system, the programmer will use the functions from the modules to create, acquire and release objects. In addition the two modules are responsible for communicating with the main DSMCBE modules which are located on the PPC.

The PPE handler is responsible for handling communication between the PPC user code and the request coordinator (see below). Like the PPE handler, the SPE handler is responsible for handling communication between user code on the SPEs and the request coordinator (see below). However the SPE handler also manages allocation and deallocation of Local Store (LS) memory, which enables the SPE handler to perform memory management without interrupting the SPEs.

The DSMCBE library uses a single processing thread, called the request coordinator, which is responsible for servicing requests from the other modules. Components can then communicate with the request coordinator by supplying a target for the answer. Using this single thread approach makes it simpler to execute atomic operations and reduces the number of locks to a pair per participating component. Each PPC thread and SPE unit functions as a single component, which results in the request coordinator being unable to determine if the participant is a PPC thread or a SPE. As most requests must pass through the request coordinator, an obvious drawback to this method is that it easily becomes a bottleneck. With this communication framework it is easier to implement channel based communication, as the *Request Coordinator* can simply be extended to handle channel requests.

2.2. Extending DSMCBE with Channel Based Communication for CELL-BE

This section will describe how we propose to extend the DSMCBE model with channel based communication. We have used the DSMCBE system as a framework to ensure atomicity and enable memory transfers within the CELL-BE processor. The implementation does not use any DSM methods and consists of a separate set of function calls.

We have intentionally made the programming model very simple; it consists of only six functions:

- `dsmcbe_csp_channel_read`
- `dsmcbe_csp_channel_write`
- `dsmcbe_csp_item_create`
- `dsmcbe_csp_item_free`
- `dsmcbe_csp_channel_create`
- `dsmcbe_csp_channel_poison`

All functions return a status code which describes the outcome of the call.

2.2.1. Channel Communication

The basic idea in the communication model is to use channels to communicate. There are two operations defined for this: `dsmcbe_csp_channel_read` and `dsmcbe_csp_channel_wri-te`. As in other CSP implementations, the read and write operations block until a matching request arrives, making the operations a synchronized atomic event.

When writing to a channel, the calling process must supply a pointer to the data area. The result of a read operation is a pointer to a data area, as well as the size of the data area. After receiving a pointer the caller is free to read and write the contents of the area. As the area is exclusively owned by the process there is no possibility of a race condition. As it is possible to write arbitrary memory locations, when using C, it is the programmers responsibility not to use the data area after a call to write. Logically, the caller can consider the `dsmcbe_csp_channel_write` operation as transferring the data and ownership of the area to the recipient. After receiving a pointer from a read operation, and possibly modifying data area, the process may forward the pointer again using `dsmcbe_csp_channel_write`. As the reading process has exclusive ownership of the data area, it is also responsible for freeing the data area, if it is no longer needed.

The operation results in the same output regardless of which CELL-BE processor the call originates from. If both processes are in the same memory space the data is not copied ensuring maximal speed. If the data requires a transfer, the library will attempt to do so in the most efficient manner.

2.2.2. Transferable Items

The CELL-BE processor requires that data is aligned and have certain block sizes, a constraint that is not normally encountered by a programmer. We have chosen to expose a simple pair of functions that mimic the well-known `malloc` and `free` functions called `dsmcbe_csp_item_create` and `dsmcbe_csp_item_free`, respectively. A process wishing to communicate can allocate a block of memory by calling the `dsmcbe_csp_item_create` function and get a standard pointer to the allocated data area. The process is then free to write data into the allocated area. After a process has used a memory block, it can either forward the block to another channel, or release the resources held by calling `dsmcbe_csp_item_free`.

2.2.3. Channel Creation

When the programmer wants to use a channel it is necessary to create it by calling the `dsmcbe_csp_channel_create` method. To distinguish channels, the create function must be called with a unique number, similar to a channel name or channel object in other CSP systems. This channel number is used to uniquely identify the channel in all subsequent communication operations.

The create function allows the caller to set a buffer size on the channel, thus allowing the channel writers to write data into the channel without awaiting a matching reader. A buffer in the CSP model works by generating a sequence of processes where each process simply reads and writes an element. The number of processes in the chain determines the size of the buffer. The semantics of the implemented buffer are the same as a chain of processes, but the implementation uses a more efficient method with a queue.

The channel type specifies the expected use of the channel, with the following options: one-to-one, one-to-any, any-to-one, any-to-any and one-to-one-simple. Using the channel type it is possible to verify that the communication patterns correspond to the intended use. In situations where the participating processes do not change it is possible to enable "low overhead" communication by using the channel type one-to-one-simple. Section 2.2.8 describes this optimization in more detail.

A special convention borrowed from the DSMCBE model is that read or write operations on non-existing channels will cause the caller to block if the channel is not yet created. Since a program must call the create function exactly once for each channel, some start-up situations are difficult to handle without this convention. Once a process has created the channel, it processes all the pending operations as if they occurred after the channel creation.

2.2.4. Channel Poison

As all calls are blocking they can complicate the shutdown phase of a CSP network. The current CSP implementations support a channel poison state, which causes all pending and following operations on that channel to return the poison.

To poison a channel, a process calls `dsmcbe_csp_channel_poison` with the id of an existing channel. When using poison, it is important to check the return value of the read and write operations, as they may return the poison status. A macro named `CSP_SAFE_CALL` can be used to check the return value and exit the current function when encountered. However the programmer is still fully responsible for making the program handle and distribute poison correctly.

2.2.5. External Choice

As a read operation is blocking, it is not possible to wait for data on more than one channel, nor is it possible to probe a channel for its content. If a process could see whether or not a channel has content, a race condition could be introduced. Thereby a second process could read the item right after the probe, resulting in a blocking read.

To solve this issue, CSP uses the concept of external choice where a process can request data from multiple channels and then gets a response once a channel is ready. To use external choice, the process must call a variation of the `dsmcbe_csp_channel_read` function named `dsmcbe_csp_channel_read_alt`, where `alt` is short for “alternation”, the term used in C.A.R. Hoare’s original paper [25]. Using this function, the process can block for a read operation on multiple channels. When one of the channels has data, the data is returned, as with the normal read operation, along with the channel id of the originating channel. This way of dealing with reads ensures that race conditions cannot occur.

With the channel selection done externally, the calling process has no way of controlling which channel to read, should there be multiple available choices. To remedy this, the calling process must also specify what strategy to use if multiple channels are ready. The JCSP library offers three strategies: arbitrary, priority and fair. Arbitrary picks a channel at random whereas priority chooses the first available channel, prioritized by the order in which the channels are given. Fair selection keeps count of the number of times each channel has been selected and attempts to even out the usage of channels. The current implementation of CSP channels for CELL-BE only supports priority select, but the programmer can emulate the two other modes.

Similar to the read function, a function called `dsmcbe_csp_channel_write_alt` allows a process to write to the first available channel. This function also supports a selection strategy and returns the id of the channel written to. There is currently no mechanism to support the simultaneous selection of channel readers and writers, though there are other ways of engineering this.

2.2.6. Guards

To prevent a call from blocking, the calling function can supply a guard which is invoked when no data is available. The implementation defines a reserved channel number, called `CSP_SKIP_GUARD` which can be given as a channel id when requesting read or write from multiple channels. If the operation would otherwise block, the function returns a `NULL` pointer and `CSP_SKIP_GUARD` as the channel value.

Other CSP implementations also offer a time-out guard, which performs a skip, but only if the call blocks for a certain period. This functionality is not available in the current implementation, but could be added without much complication.

2.2.7. Processes for CELL-BE

The hardware in the CELL-BE is limited to a relatively low number of physical SPEs, which prevents the generation of a large number of CSP processes. To remedy this situation the implementation also supports running multiple processes on each SPE. Since the SPEs have little support for timed interrupts, the implementation is purely based on cooperative switching. To allow multiple processes on the SPE, we have used an approach similar to CELL-MT [26], basically implementing a user-mode thread library, but based on the standard C functions `setjmp` and `longjmp`.

The CSP threading library implements the `main` function, and allocates ABI compliant stacks for each of the processes when started. After setting up the multithreading environment, the scheduler is activated which transfers control to the first processes. Since the `main` function is implemented by the library, the user code must instead implement the `dsmcbe_main` function, which is activated for each process in turn. This means that all processes running on a single SPE must use the same `dsmcbe_main` function, but each process can call the function `dsmcbe_thread_current_id` and thus obtain a unique id, which can be used to determine what code the process will execute.

When a process is executing it can cooperatively yield control by calling `dsmcbe_thread_yield`, which will save the process state and transfer control to the next available process. Whenever a process is waiting for an API response, the library will automatically call a similar function called `dsmcbe_thread_yield_ready`. This function will yield if another process is ready to execute, meaning that it is not currently awaiting an API response. The effect of this is that each API call appears to be blocking, allowing the programmer to write a fully sequential program and transparently run multiple processes.

As there is no preemptive scheduling of threads, it is possible for a single process to prevent other processes from executing. This is a common trade-off between allowing the SPE to execute code at full speed, and ensuring progress in all processes. This can be remedied by inserting calls to `dsmcbe_thread_yield_ready` inside computationally heavy code, which allows the programmer to balance the single process execution and overall system progress in a fine grained manner.

The scheduler is a simple round-robin scheduler using a ready queue and a waiting queue. The number of threads possible is limited primarily by the amount of available LS memory, which is shared among program code, stack and data. The running time of the scheduler is $O(N)$ which we deem sufficient, given that all processes share the limited LS, making more than 8 processes per SPE unrealistic.

2.2.8. SPE-to-SPE Communication

Since the PPC is rarely a part of the actual problem solving, the memory blocks can often be transferred directly from SPE to SPE without transferring it into main memory.

If a SPE is writing to a buffered channel, the data may not be read immediately after the write. Thus, the SPE may run out of memory since the data is kept on the SPE in anticipation of a SPE-to-SPE transfer. To remedy this, the library will flush data to main memory if an allocation would fail. This is in effect a caching system, and as such it is subject to the regular benefits and drawbacks of a cache. One noticeable drawback is that due to the limited available memory, the SPEs are especially prone to memory fragmentation, which happens more often when using a cache, as the memory stays fully populated for longer periods.

If the channel is created with the type one-to-one-simple, the first communication will be used to determine the most efficient communication pattern, and thus remove some of the internal synchronization required. If two separate SPEs are communicating, this means that the communication will be handled locally in the *SPE Handler* shown in Figure 1, and thus eliminate the need to pass messages through the *Request Coordinator*.

A similar optimization is employed if two processes on the same SPE communicate. In this case the data is kept on the SPE, and all communication is handled locally on the SPE in the *DSMCBE SPE* module shown in Figure 1. Due to the limited amount of memory available on the SPE, data may be flushed out if the channel has large buffers or otherwise exhaust the available memory.

These optimizations can only work if the communication is done in a one-to-one fashion where the participating processes never change. Should the user code attempt to use such a channel in an unsupported manner, an error code will be returned.

2.2.9. Examples

To illustrate the usage of the channel-based communication Listing 1 shows four simple CSP processes. Listing 2 presents a simple example that uses the alternation method to read two channels and writes the sum to an output channel.

3. Experiments

When evaluating system performance, we focus mainly on the scalability aspect. If the system scales well, further optimizations may be made specific to the application, utilizing the SIMD capabilities of the SPEs. The source code for the experiments are available from `http://code.google.com/p/dsmcbe/`.

```
#include <dsmcbe_csp.h>

int delta1(GUID in, GUID out) {
  void* value;

  while(1) {
    CSP_SAFE_CALL("read", dsmcbe_csp_channel_read(in, NULL, &value));
    CSP_SAFE_CALL("write", dsmcbe_csp_channel_write(out, value));
  }
}

int delta2(GUID in, GUID outA, GUID outB) {
  void* inValue, outValue;
  size_t size;

  while(1) {
    CSP_SAFE_CALL("read", dsmcbe_csp_channel_read(in, &size, &inValue));
    CSP_SAFE_CALL("allocate", dsmcbe_csp_item_create(&outValue, size));

    memcpy(outValue, inValue, size); //Copy contents as we need two copies

    CSP_SAFE_CALL("write A", dsmcbe_csp_channel_write(outA, inValue));
    CSP_SAFE_CALL("write B", dsmcbe_csp_channel_write(outB, outValue));
  }
}


int prefix(GUID in, GUID out, void* data) {
  CSP_SAFE_CALL("write", dsmcbe_csp_channel_write(out, data));

  return delta1(in, out);
}

int tail(GUID in, GUID out) {
  void* tmp;

  CSP_SAFE_CALL("read", dsmcbe_csp_channel_read(in, NULL, &tmp));
  CSP_SAFE_CALL("free", dsmcbe_csp_item_free(tmp));

  return delta1(in, out);
}
```

Listing 1. Four simple CSP processes.

```
int add(GUID inA, GUID inB, GUID out)
{
  void *data1, *data2;

  GUID channelList[2];
  channelList[0] = inA;
  channelList[1] = inB;

  GUID chan;

  while(1)
  {
    dsmcbe_csp_channel_read_alt(CSP_ALT_MODE_PRIORITY, channelList, 2, &chan,
        NULL, &data1);
    dsmcbe_csp_channel_read(chan == inA ? inB : inA, NULL, &data2);

    *(int*)data1 = *((int*)data1) + *((int*)data2);

    dsmcbe_csp_item_free(data2);
    dsmcbe_csp_channel_write(out, data1);
  }
}
```

Listing 2. Reading from two channels with alternation read and external choice. To better fit the layout of the article the `CSP_SAFE_CALL` macro is omitted.

All experiments were performed on an IBM QS22 blade, which contains 2 connected CELL-BE processors, giving access to 4 PPE cores and 16 SPEs.

3.1. CommsTime

A common benchmark for any CSP implementation is the CommsTime application which sets up a ring of processes that simply forwards a single message. The conceptual setup is shown in Figure 2. This benchmark measures the communication overhead of the channel operations since there is almost no computation required in the processes. To better measure the scalability of the system, we have deviated slightly from the normal CommsTime implementation, by inserting extra successor processes as needed. This means that each extra participating process will add an extra channel, and thus and thus produce a longer communication ring.

Figure 3 shows the CommsTime when communicating among SPE processes. The PPE records the time between each received message, thus measuring the time it takes for the message to traverse the ring. The time shown is an average over 10 runs of 10.000 iterations. As can be seen, the times seems to stabilize around 80 μseconds when using one thread per SPE. When using two or more threads the times stabilizes around 38 μseconds, 27 μseconds, and 20 μseconds respectively. When using multiple threads, the communication is performed internally on the SPEs, which results in a minimal communication overhead causing the average communication overhead to decrease.

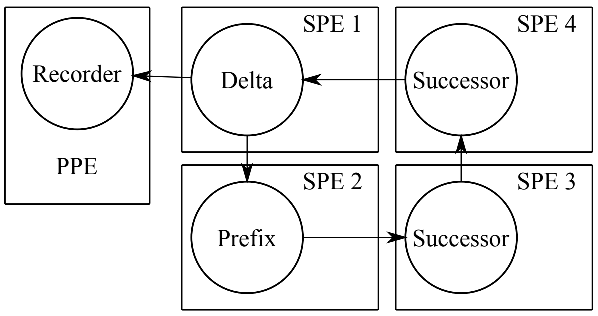


Figure 2. Conceptual setup for the CommsTime experiment with 4 SPEs.

We have executed the CommsTime sample from the JCSP library v.1.1rc4 on the PPE. The JCSP sample uses four processes in a setup similar to Figure 2 but with all processes placed on the PPE. Each communication took on average 63 μseconds which is slightly faster than our implementation, which runs at 145 μseconds on the PPE. Even though JCSP is faster, it does not utilize the SPEs, and cannot utilize the full potential of the CELL-BE.

3.2. Prototein Folding

Prototeins are a simplified 2D model of a protein, with only two amino acids and only 90 degree folds [27]. Folding a prototein is computationally simpler than folding a full protein, but exhibit the same computational characteristics. Prototein folding can be implemented with a bag-of-tasks type solution, illustrated in Figure 4, where partially folded prototeins are placed in the bag. The partially folded prototeins have no interdependencies, but may differ in required number of combinations and thus required computational time.

As seen in Figure 5 the problem scales very close to linearly with the number of SPEs, which is to be expected for this type of problem. This indicates that the communication latency is not a limiting factor, which also explains why the number of SPE threads have very little effect on the scalability.

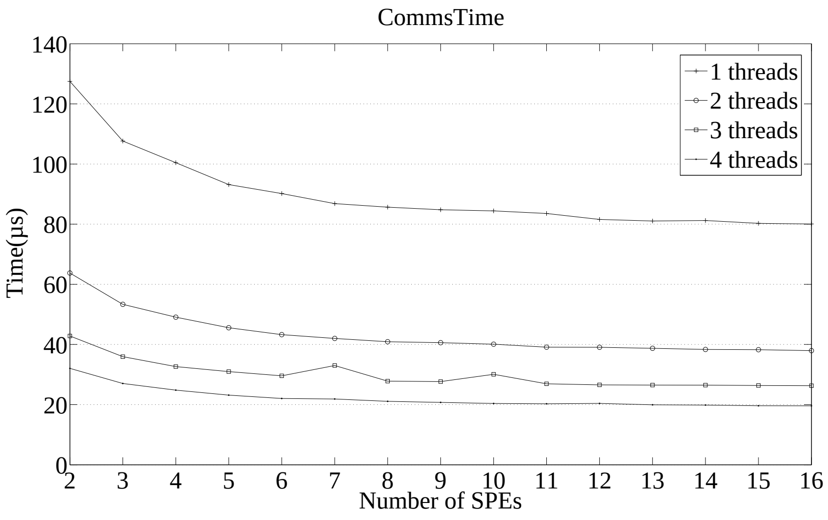


Figure 3. CommsTime using 2-16 SPEs with 1-4 threads per SPE.

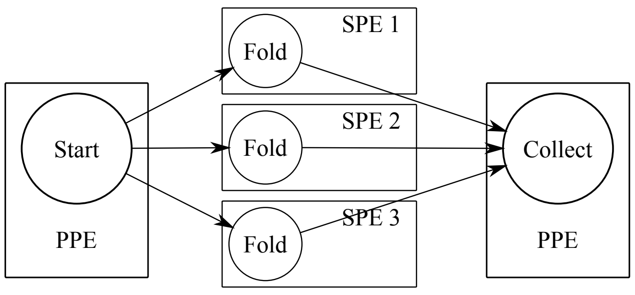


Figure 4. Conceptual setup for Prototein folding with 3 SPEs.

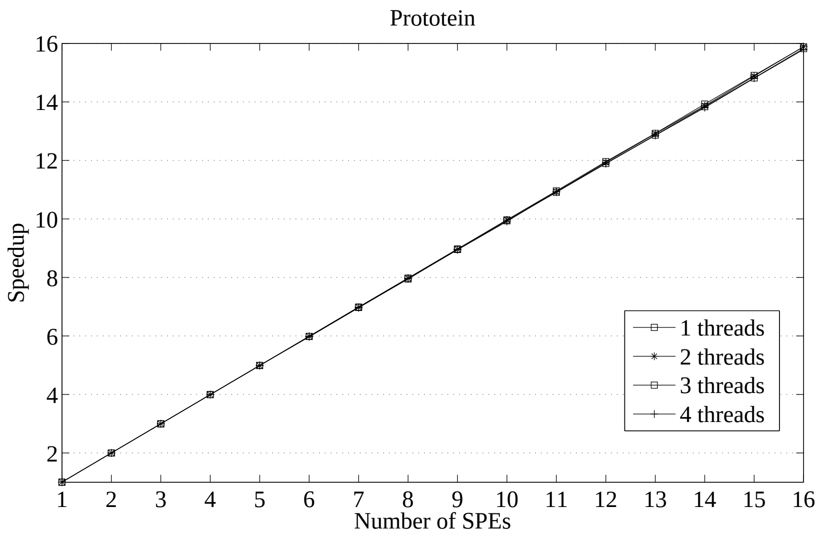


Figure 5. Speedup of prototein folding using 1-16 SPEs.

3.3. k Nearest Neighbors (kNN)

The kNN application is a port of a similar application written for PyCSP [28]. Where the PyCSP model is capable of handling an extreme number of concurrent processes, the library is limited by the number of available SPEs and the amount of threads each SPE can accommodate. Due to this, the source code for the two applications are hard to compare, but the overall approach and communication patterns are the same. Figure 6 shows a conceptual ring based setup for finding the kNN.

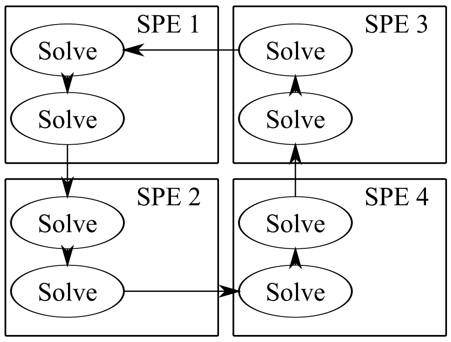


Figure 6. Conceptual setup for the kNN experiment with 4 SPEs, each running 2 threads.

This ring-based approach means that each process communicates only with its neighbor. To support arbitrary size problems, one of the channels are buffered. The underlying system will attempt to keep data on the SPE, in anticipation of a transfer, but as the SPE runs out of memory, the data will be swapped to main memory. This happens completely transparent to the process, but adds an unpredictable overhead to the communication. This construction allows us to run the same problem size on one to 16 SPEs.

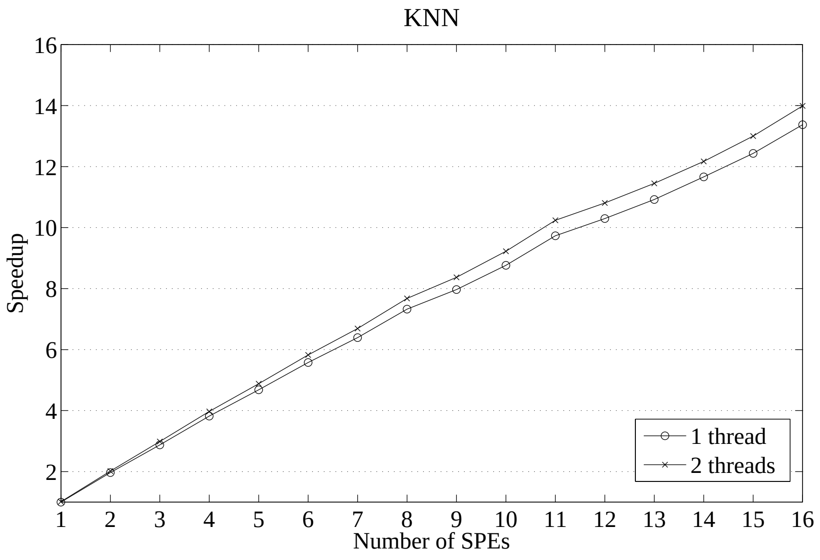


Figure 7. Speedup of the k Nearest Neighbors problem using 1-16 SPEs to search for 10 nearest neighbors in a set with 50k elements with 72 dimensions.

As seen in Figure 7 this does not scale linearly, but given the interdependencies we consider this to be a fairly good result. Figure 7 also shows that using threads to run multiple solver processes on each SPE offers a performance gain, even though the processes compete for the limited LS memory. This happens because the threads implement an implicit form of double buffering, allowing each SPE to mask communication delays with computation. The achieved speedup indicates that there is a good balance between the communication and computation performed in the experiment.

The speedup for both graphs is calculated based on the measured time for running the same problem size on a single SPE with a single solver thread.

3.4. Communication to Computation Ratio

The ring based communication model used in the kNN experiment is quite common for problems that use a n^2 approach. However, the scalability of such a setup is highly dependent on the amount of work required in each subtask. To quantify the communication to computation ratio required for a well-scaling system, we have developed a simple ring-based program that allows us to adjust the number of floating point operations performed between communications. The computation performed is adjustable and does not depend on the size of the transmitted data, allowing us to freely experiment with the computational workload. The setup for this communication system is shown in Figure 8. The setup is identical to the one used in the kNN experiment, but instead of having two communicating processes on the same SPE, the processes are spread out. This change cause the setup to loose the possibility for the very fast internal SPE communication channels, which causes more load on the PPE and thus gives a more realistic measurement for the communication delays.

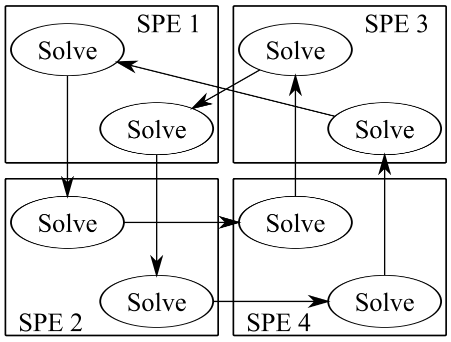


Figure 8. Conceptual setup for non-structured ring based communication.

As seen in Figure 9, the implementation scales well if computation performed in each ring iteration is around 100MFLOPS. Comparing the two graphs in Figure 9, shows that increasing the number of threads on the SPEs, results in a decrease in performance. This happens because the extra processes introduce more communication. This increase in communication causes a bigger strain on the PPE, which results in more latency than the processes hide. In other words, the threads cause more latency than they can hide in this setup.

The speedup for both graphs in Figure 9 are calculated based on measurements from a run with the same data size on a single SPE with a single thread.

Comparing the Communication to Computation experiment with the kNN experiment reveals that the use of optimized channels reduces the latency of requests to a level where the threads are unable to hide the remaining latency. In other words, the latency becomes so low, that the thread switching overhead is larger than the latency it attempts to hide. This is consistent with the results from the CommsTime experiment, which reveals that the communication time is very low when performing inter-SPE communication. This does not mean that the latency is as low as it can be, but it means that the extra communication generated by the threads increases the amount of latency that must be hidden.

4. Future Work

The main problem with any communication system is the overhead introduced by the communication. As the experiments show, this overhead exists but can be hidden because the CELL-BE and library are capable of performing the communication and computation simultaneously. But this hiding only works if the computational part of a program has a sufficient size. To remedy this, the communication overhead should be reduced significantly.

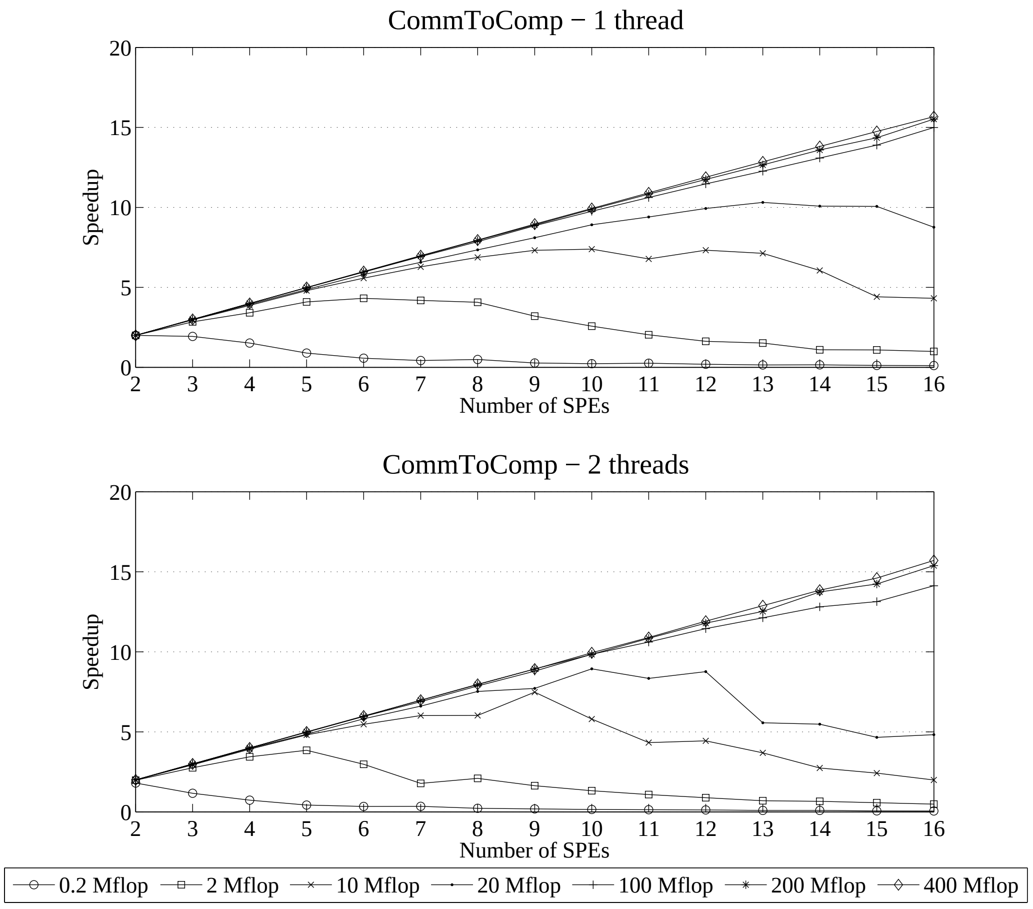


Figure 9. Communication To Computation ratio, 16 bytes of data.

The decision to use the request coordinator to handle the synchronization simplifies the implementation, but also introduces two performance problems. One problem is that if the system becomes overwhelmed with requests, the execution will be sequential, as the processes will only progress as fast as the request coordinator responds to messages. The other problem is that the requests pass through both the SPU handler and the request coordinator, which adds load to the system and latency to each communication operation.

4.1. Reduce Request Latency

Since the SPEs are the main workhorse of the CELL-BE, it makes sense to move much of the decision logic into the SPU handler rather than handle it in the request coordinator. The request coordinator is a legacy item from the DSM system, but there is nothing that prevents participating PPE processes from communicating directly with the SPU handler.

4.2. Increase Parallelism

Even if the request coordinator is removed completely, the PPE can still be overwhelmed with requests, which will make everything run sequentially rather than in parallel. It is not possible to completely remove a single synchronization point, but many communication operations involve exactly two processes. In the common case where these two processes reside on separate SPEs, it is possible to perform direct SPE-to-SPE communication through the use of signals and DMA transfers. If this is implemented, it will greatly reduce the load on the PPE for all the presented experiments.

4.3. Improve Performance of the SPU Handler

The current implementation uses a shared spinning thread that constantly checks for SPE and request coordinator messages. It is quite possible that this can be improved by using a thread for each SPE which uses the SPE events rather than spinning. Experiments performed for the DSMCBE [16] system show that improving the SPU handler can improve the overall system performance.

4.4. Improve Memory Exhaustion Handling

When the communication is handled by the SPEs internally, it is likely that they will run out of memory. If the SPU handler is involved, such situations are detected and handled gracefully. Since this is essentially a cache system, a cache policy can greatly improve the performance of the system, by selectively choosing which elements to remove from the LS and when such an operation is initiated.

4.5. Process Migration

The processes are currently bound to the SPE that started them, but it may turn out that the setup is ineffective and can be improved by moving communicating processes closer together, i.e. to the same SPE. There is limited support for this in the CELL-BE architecture itself, but the process state can be encapsulated to involve only the current thread stack and active objects. However, it may prove to be impossible to move a process, as data may occupy the same LS area. Since the C language uses pointers, the data locations cannot be changed during a switch from one SPE to another. One solution to this could be to allocate processes in *slots*, such as those used in CELL CSP [17].

4.6. Multiple Machines

The DSMCBE system already supports multiple machines, using standard TCP-IP communication. It would be desirable to also support multiple machines for CSP. The main challenge with multiple machines is to implement a well-scaling version of the alternation operations, because the involved channels can span multiple machines. This could use the cross-bar approach used in JCSP [29].

5. Conclusion

In this paper we have described a CSP inspired communication model and a thread library, that can help programmers handle the complex programming model on the CELL-BE. We have shown that even though the presented models introduce some overhead, it is possible to get good speedup for most problems. On the other hand Figure 9 shows, that if the computation to communication ratio is too low - meaning too little computation per communication, it is very hard to scale the problems to utilize all 16 SPEs. However we believe that for most programmers solving reasonable sized problems, the tools provided can significantly simplify the writing of programs for the CELL-BE architecture.

We have also shown that threads can be used to mask some latency, but at the same time they generate some latency, which limits their usefulness to certain problems.

DSMCBE and the communication model described in this paper is open source software under the LGPL license, and are available from `http://code.google.com/p/dsmcbe/`.

Acknowledgements

The authors acknowledge the Danish National Advanced Technology Foundation (grant number 09-067060) and the innovation consortium (grant number 09-052139) for supporting this research project. Furthermore the authors acknowledge Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation, for the use of Cell Broadband Engine resources that have contributed to this research.

References

[1] Wm. A. Wulf and Sally A. Mckee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23:20–24, 1995.

[2] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[3] Gordon E. Moore. Readings in computer architecture. chapter Cramming more components onto integrated circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[4] Thomas Chen. Cell Broadband Engine Architecture and its first implementation - A Performance View, 2005. `http://www.ibm.com/developerworks/power/library/pa-cellperf/`. Accessed 26 July 2010.

[5] Martin Rehr. Application Porting and Tuning on the Cell-BE Processor, 2008. `http://dk.migrid.org/public/doc/published_papers/nqueens.pdf`. Accessed 26 July 2010.

[6] Mohammed Jowkar. Exploring the Potential of the Cell Processor for High Performance Computing, 2007. `http://www.diku.dk/~rehr/cell/docs/mohammad_jowkar_thesis.pdf`. Accessed 26 July 2010.

[7] IBM. IBM Doubles Down on Cell Blade, 2007. `http://www-03.ibm.com/press/us/en/pressrelease/22258.wss`. Accessed 26 July 2010.

[8] IBM. Cell BE Programming Handbook Including PowerXCell 8i, 2008. `https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1741C509C5F64%B3300257460006FD68D/$file/CellBE_PXCell_Handbook_v1.11_12May08_pub.pdf`. Accessed 26 July 2010.

[9] Jakub Kurzak, Alfredo Buttari, and Jack Dongarra. Solving Systems of Linear Equations on the CELL Processor Using Cholesky Factorization. *IEEE Trans. Parallel Distrib. Syst.*, 19(9):1175–1186, 2008.

[10] Asim Munawar, Mohamed Wahib, Masaharu Munetomo, and Kiyoshi Akama. Solving Large Instances of Capacitated Vehicle Routing Problem over Cell BE. In *HPCC '08: Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications*, pages 131–138, Washington, DC, USA, 2008. IEEE Computer Society.

[11] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-131-53271-5.

[12] A.W. Roscoe, C.A.R. Hoare, and R. Bird. *The theory and practice of concurrency*, volume 216. Citeseer, 1998.

[13] IBM. Accelerated Library Framework Programmer's Guide and API Reference, 2009. `http://public.dhe.ibm.com/software/dw/cell/ALF_Prog_Guide_API_v3.1.pdf`. Accessed 26 July 2010.

[14] Kevin O'Brien, Kathryn O'Brien, Zehra Sura, Tong Chen, and Tao Zhang. Supporting OpenMP on Cell. In *IWOMP '07: Proceedings of the 3rd international workshop on OpenMP*, pages 65–76, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. CellSs: a Programming Model for the Cell BE Architecture. In *ACM/IEEE CONFERENCE ON SUPERCOMPUTING*, page 86. ACM, 2006.

[16] Morten N. Larsen, Kenneth Skovhede, and Brian Vinter. Distributed Shared Memory for the Cell Broadband Engine (DSMCBE). In *ISPDC '09: Proceedings of the 2009 Eighth International Symposium on Parallel and Distributed Computing*, pages 121–124, Washington, DC, USA, 2009. IEEE Computer Society.

[17] Mads Alhof Kristiansen. CELL CSP Sourcecode, 2009. `http://code.google.com/p/cellcsp`. Accessed 26 July 2010.

[18] Christian L. Jacobsen and Matthew C. Jadud. The Transterpreter: A Transputer Interpreter. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, volume 62 of *Concurrent Systems Engineering Series*, pages 99–106, Amsterdam, September 2004. IOS Press.

[19] Damian J. Dimmich, Christian L. Jacobsen, and Matthew C. Jadud. A Cell Transterpreter. In Peter Welch, Jon Kerridge, and Fred Barnes, editors, *Communicating Process Architectures 2006*, volume 29 of *Concurrent Systems Engineering Series*, pages 215–224, Amsterdam, September 2006. IOS Press.

[20] Ulrik Schou Jørgensen and Espen Suenson. trancell - an Experimental ETC to Cell BE Translator. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 287–298, jul 2007.

[21] Alistair A. Mcewan, Steve Schneider, Wilson Ifill, Peter Welch, and Neil Brown. C++CSP2: A Many-to-Many Threading Model for Multicore Architectures, 2007.

[22] P. H. Welch, A. W. P. Bakkers (eds, and Nan C. Schaller. Using Java for Parallel Computing - JCSP versus CTJ. In *Communicating Process Architectures 2000*, pages 205–226, 2000.

[23] Otto J. Anshus, John Markus Bjørndalen, and Brian Vinter. PyCSP - Communicating Sequential Processes for Python. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 229–248, jul 2007.

[24] Brian Vinter, John Markus Bjørndaln, and Rune Møllegaard Friborg. PyCSP Revisited, 2009. `http://pycsp.googlecode.com/files/paper-01.pdf`. Accessed 26 July 2010.

[25] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[26] Vicenç Beltran, David Carrera, Jordi Torres, and Eduard Ayguadé. CellMT: A cooperative multithreading library for the Cell/B.E. In *HiPC*, pages 245–253, 2009.

[27] Brian Hayes. Prototeins. *American Scientist*, 86(3):216–, 1998.

[28] Rune Møllegaard Friborg. PyCSP kNN implementation, 2010. `http://pycsp.googlecode.com/svn-history/r288/trunk/examples/kNN.py`. Accessed 26 July 2010.

[29] P.H. Welch and B. Vinter. Cluster Computing and JCSP Networking. *Communicating Process Architectures 2002*, 60:203–222, 2002.

A Journal of Software Engineering and Applications, **2012, *, **-****
doi:10.4236/jsea.2012.***** Published Online *** 2012 (http://www.scirp.org/journal/jsea)

A Distributed Virtual Machine for Microsoft .NET

Morten N. Larsen and Brian Vinter

The Niels Bohr Institute, University of Copenhagen, Copenhagen, Denmark
Email: momi@nbi.ku.dk, vinter@nbi.ku.dk

Received October 24, 2012.

ABSTRACT

Today, an ever increasing number of natural scientists use computers for data analysis, modeling, simulation and visualization of complex problems. However, in the last decade the computer architecture has changed significantly, making it increasingly difficult to fully utilize the power of the processor, unless the scientist is a trained programmer. The reasons for this shift include the change from single-core to multi-core processors, as well as the decreasing price of hardware, which allows researchers to build cluster computers made from commodity hardware. Therefore, scientists must not only be able to handle multi-core processors, but also the problems associated with writing distributed memory programs and handle communication between hundreds of multi-core machines. Fortunately, there are a number of systems to help the scientist e.g. Message Parsing Interface (MPI) [1] for handling communication, DistNumPy [2] for handling data distribution and Communicating Sequential Processes (CSP) [3] for handling concurrency related problems. Having said that, it must be emphasized that all of these methods require that the scientists learn a new method and then rewrite their programs, which mean more work for the scientist. A solution that does not require much work for the scientists is automatic parallelization. However, research dating back three decades has yet to find fully automated parallelization as a feasible solution for programs in general, but some classes of programs can be automatically parallelized to an extent.

This paper describes an external library which provides a `Parallel.For` loop construct, allowing the body of a loop to be run in parallel across multiple networked machines, i.e. on distributed memory architectures. The individual machines themselves may be shared memory nodes of course. The idea is inspired by Microsoft's Parallel Library that supplies multiple `Parallel` constructs. However, unlike Microsoft's Library our library supports distributed memory architectures. Preliminary tests have shown that simple problems may be distributed easily and achieve good scalability. Unfortunately, the tests show that the scalability is limited by the number of accesses made to shared variables. Thus the applicability of the library is not general but limited to a subset of applications with only limited communication needs.

Keywords: *Microsoft .NET, Parallelization, Distribution*, Data parallelism

1. Introduction

During the last decade the usage of high performance computing has increased beyond classic areas for scientific computing, the type of problems that are solved by high performance computing has widen, but most importantly the user group has changed from programming specialist to a more mixed group of scientists from fields like chemistry, physics, environmental sciences, engineering etc. These two factors have meant that the tools for aiding the users in handling hardware are more important today than ever before. As a natural consequence, there is an increase in the solutions that can help the users. Solutions ranging from automatic parallelization to tools like Message Parsing Inteface (MPI) and Communicating Sequential Processes (CSP). Nevertheless, many of the tools available have very little usage in practice and/or do not provide enough scalability compared to the manually written code. However, the greatest problem is that many of the tools have a very steep learning curve, and thus, presents problems for many non-computer specialists, who may be able to write a sequential program, but do not have knowledge of locks, raise conditions, deadlocks and memory layout.

In an attempt circumvent this problem Microsoft has in recent years improved .NET with tools to help users write parallel code. The functionality resides mainly in the Microsoft Parallel Library [4] and consists of a set of tools; however, this paper focus exclusively on one, namely the `Parallel.For` construct. The construct as the name reveals, is the parallel version of the normal `For`-loop. The usage is very simple and the users should in theory just replace the `For`-loops with the `Parallel.For` loop and the code will then be executed across all available cores in the machine. Importantly though; in the current version of the tool the parallelization does not go beyond a single shared memory machine.

To improve Microsoft's idea by enabling distribution beyond a single machine, we have examined Microsoft .NET and the Microsoft Parallel system and will in this paper describe a solution for adding an external module to the system. The focus has been on making minimal changes to the code compared to the original code with a `Parallel.For` loop. Furthermore, the use of Microsoft.Parallel has been replaced by our implementation named DistVES (Distributed Virtual Execution System) as described in this work. From the beginning it was clear that the proposed solution would not work for every type of .NET program especially not programs with many interrupts, GUI programs, programs that have a lot of disk usage, etc. Therefore, the target programs have been limited to scientific application e.g. data analysis, modeling, simulation and visualization. Furthermore, simple algorithms which should yield good speed-up have been chosen for testing the initial version.

The rest of the paper is structured as follows: section 2 gives a short introduction to Microsoft's Common Intermediate Language, which is the level at which DistVES transforms the original code. Section 3 gives a description of the design including consistency, client/server and code generation. In section 4 the results of running a number of benchmarks are discussed. Future work is described in section 5 and finally section 6 gives a summary of our findings.

1.1. Related work

DistVES is as mentioned above, closely related to Microsoft Parallel Library with the main difference that DistVES supports multiple machines. This clearly changes the intrinsic properties of the two systems, but for the users the two systems seem similar. Another closely related system is OpenMP [5] which needs to be incorporated in the compiler of a given programming language and many C/C++ and Fortran based programming languages are supported including .NETs Visual C++. Originally, OpenMP only supported shared-memory multiprocessor platforms, but IBM has orked on a version that supports a cluster [6]. Yet another way to help the programmer is to have support for distributed shared memory on the .NET objects. However, due to problems with scalability and usability, these types of systems have never proved a good solution [7]. Common for the three methods is that they only result in good scalability when the implemented algorithms are very simple and straightforward to parallelize.

A lot of research over the last decades has been dedicated to auto-parallelization. The general position is that it only works for very simplified algorithms and therefore alternative solutions must be found. Instead of auto-parallelization systems, some systems focus on making the communication between machines easier. Systems like the MPI provide functionality to distribute and run tasks on a large set of computers and gather the results of the computations. Likewise systems of the CSP type provide mechanisms of communication between different machines. The goal of CSP is to help the programmer writing correct code e.g. free of live-locks, dead-locks, and race conditions.

Ultimately, before most scientists can fully utilize large parallel machines, it might be that a whole new approach for making hardware and new parallel programming languages must be defined [8].

2. The Common Intermediate Language (CIL)

Before describing the design of DistVES, we will give a short introduction to the Common Intermediate Language (CIL) as the language is not commonly known. CIL is the backbone of the .NET framework and is a stack-based; platform neutral and type safe object oriented assembly language designed for .NET. The purpose of CIL is to allow multiple source-languages e.g. C#, VB.NET, and F# to be compiled into the same non-platform specific assembly language. The .NET runtime can then at runtime compile the CIL assembly to a machine specific machine code. This firstly allows for cross platform usage and secondly that programs written in e.g. C# can call methods from libraries written in languages like F# or VB.NET. Figure 1 gives an overview of the pipeline from source language to machine code.

3. Design

The design of DistVES consists of three components; a distribution model, a client/server model, and a code generator. These all play a role in turning a .NET program with a `Parallel.For` construct into a distributed program that can be executed on a cluster computer. The shared fields play a key role in the system, as they should be identified in the original .NET program and made into distributed variables. Thereby making them available to all the clients in the system. Furthermore, the coherence model should ensure that the clients always see the current version of a shared variable.

3.1. Distribution including server/client

We start by giving an overview of the model and then go through the details about data coherence and code generation.

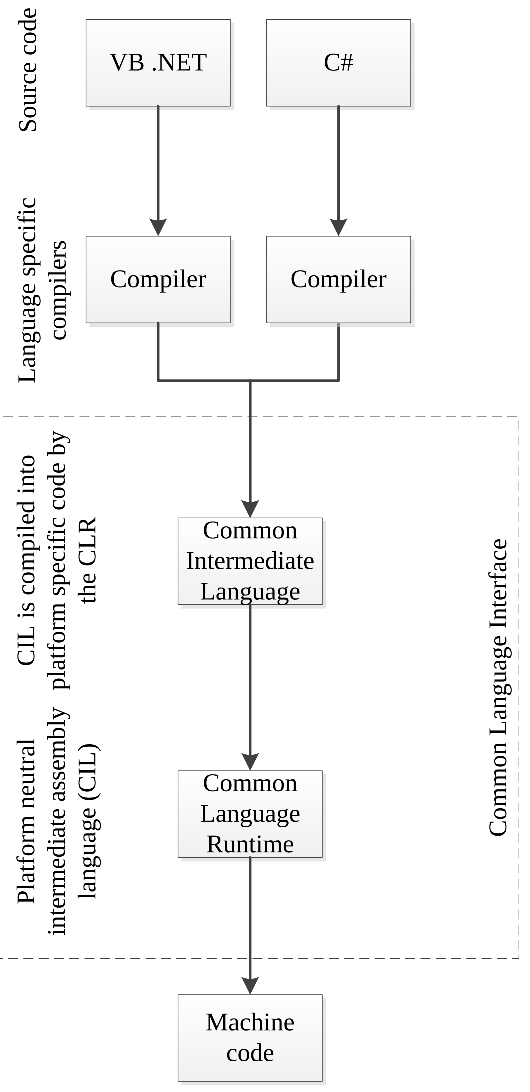

Source code
VB .NET
C#
Language specific compilers
Compiler
Compiler
CIL is compiled into platform specific code by the CLR
Platform neutral intermediate assembly language (CIL)
Common Intermediate Language
Common Language Runtime
Common Language Interface
Machine code

Figure 1. Overview of the CIL pipeline

For simplicity of implementation a central server model without client-to-client communication has been chosen for the initial version. This naturally sets an upper limit of scalability, but it will be able to show if our idea has potential. Each node in the system runs a thread which is dedicated for server communication. Again for simplicity, a single machine running multiple workers still runs one communication thread per worker, even though the workers could share a single communication thread.

Sending messages over a network requires that the objects are serialized before being sent and then deserialized at the receiving end. .NET supports automatic serialization of a class when marked with the *Serializable* attribute. Many of the built-in types in .NET are marked with this attribute, but when a programmer makes a new type it is not by default serializable. Therefore, DistVES only allow the use of the primitive types e.g. int, double, float, char and single/multi-dimension arrays of primitive types which all are serializable.

When distributing a .NET program with a Parallel construct the compiler generates an action delegate (subclass to the caller class) which contains the code from inside the Parallel construct. This is unfortunately not clear from the source code and means that some local variables can be promoted to a field in the delegate (see Table 1). Furthermore, the delegate will hold a reference to the caller class. During a normal run this reference is somewhere in the local memory and may be accessed from multiple threads, but when the program is being distributed, this reference can point to a memory location on another machine. As we cannot make a deep copy, because the class may possible not be *Serializable*, every client must create a local copy that mirrors the original. At the same time a given field must have the same unique identifier in all local copies of a given class. Through this process, DistVES can ensure that updates made to one field will be distributed to all clients. In practice, this is done by having all clients register all fields using the class ID and the field name with the server when executing the constructor of a given class. The server will then return the fields unique ID, which will be used for the rest of the execution.

3.2. Data Consistency

Maintaining multiple copies of the same object on different machines requires a system to ensure data consistency, so that all machines see the same version of the data like on a conventional shared memory machine. However, having systems with latency and transfer time means that we cannot guarantee at any given point that all machines have the exact same version of an object. Nevertheless, we can guarantee that all machines at some point will get the most recent version of the object. This is called sequentially consistency [9]. More relaxed consistency models exist [10], but in order to utilize them information about access patterns is required. As the CIL assembly does not contain information about access patterns, the programmers need to annotate the source code to use a more released system. However, making the programmers annotate the code is in conflict with the goal of making it easier for the programmer to utilize distributed computers. An implementation of sequential consistency could be the MESI protocol [11], which is known from hardware cache implementations. The MESI protocol relies on an object in cache at a given time

Table 1 – Source code example, followed by the assembly view of the compiled code (the CIL instructions are omitted)

```
class Test
{
    // The following variable is a private field in the Test class
    int globalValue = 0;

    public void Run()
    {
        // The following local variable in the class Test is "promoted"
        // to a field in the delegate because it is accessed within the delegate
        int value = 0;

        /*
         * Code inside the Paralle.For loop is compiled to a subclass of
         * Test (a delegate). If the body of the Parallel.For did not touch
         * local variables in the Run method, the body of Paralle.For would
         * be compiled to a method in the Test class
         */
        Parallel.For(0, 10, i =>
            {
                int count = i;
                value = count;
                globalValue = count;
            }
        );
        // Next line gives compiler error, because the variable count is out of scope
        // value = count;

        // This should print "Value is 9 and 9"
        Console.WriteLine("Value is {0} and {1}", value, globalValue);
    }
}
```

```
___[MOD] …CodeExample.exe
   |        M A N I F E S T
   |___[NAMESPACE] CodeExample
   |   |___[CLASS] CodeExample.Test
   |   |   |      .class private auto ansi beforefieldinit
   |   |   |___[CLASS] <>c__DisplayClass1
   |   |   |   |      .class nested private auto ansi sealed beforefieldinit
   |   |   |   |      .custom instance void [mscorlib]System.Runtime…
   |   |   |   |___[FIELD] <>4__this : public class CodeExample.Test
   |   |   |   |___[FIELD] value : public int32
   |   |   |   |___[METHOD] .ctor : void()
   |   |   |   |      <Run>b__0 : void(int32)
   |   |   |
   |   |   |___[FIELD] globalValue : private int32
   |   |   |___[METHOD] .ctor : void()
   |   |   |___[METHOD] Run : void()
```

having one of four states Modified, Exclusive, Shared or Invalid. The state of an object can change over time depending on either local or remote (other caches) making changes to the object. As seen in Figure 2 the state of an object changes whenever an action is made to the object.

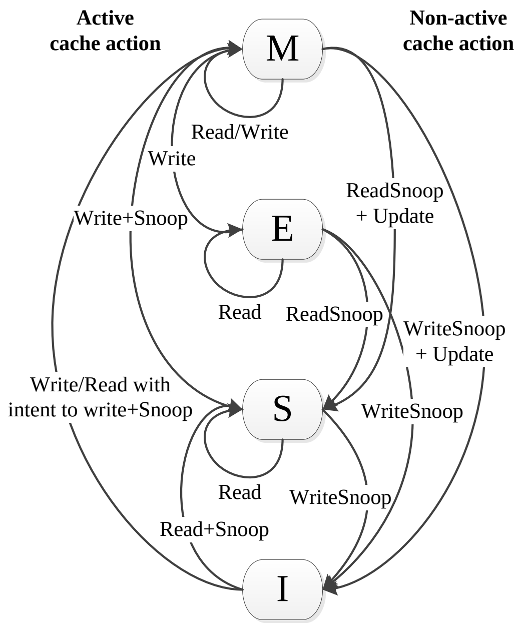
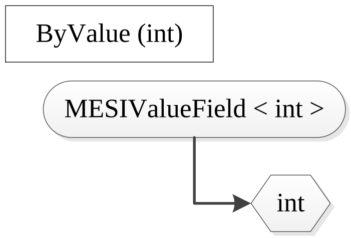


Figure 2. State transactions in the MESI protocol

Non-active cache actions are operations made by a remote cache, whereas active cache actions are operations done by the local cache. Snoop and Update actions are in the MESI model broadcasted to all other nodes in the system. Snoop broadcasts always include a type which is either "Write" or "Read" depending on how the shared object is accessed by the cache.
The next question is how to integrate the MESI protocol to an object in .NET. The most obvious way is to encapsulate all objects (those from shared fields) into a custom DistVES object which contains both the original object and the control code to acquire the functionality of the MESI protocol.
The first task in designing the custom object is to define the methods that are required to have a correctly working MESI protocol. Firstly, it should be possible to access (write/read) the original object inside the custom object. These methods are called from the user code but are blocking if the custom object's MESI state requires that the server must be contacted e.g. for an updated version of the data. Secondly, the MESI protocol requires that it is possible to "remotely" snoop the object along with the possibility to "remotely" update the object. These two methods are called from the communication thread and if the update method is called it releases the blocking user code. This is typically done in a situation where the user



code accesses an object with MESI state "invalid". Then the server is asked for an updated version of the data and the user code is blocking while waiting for a response.

Figure 3. A shared "ByValue" field encapsulated in a MESI object.

The response is handled by the communication thread and will update the data before requesting the blocking user code to continue work. Furthermore, the communication thread should handle snoop request, which mainly involves changing the MESI state of objects and/or sending an updated version of data to the server.
Now that the custom object can handle the MESI protocol, the next step is to define how the object should integrate the different types that a field can have. The shared fields in the user code can be divided into two types; value-type and reference-type. Value-type fields have the value encapsulated into the field, whereas reference-type contains a reference to an object. This yields two different implementations of the custom object as the MESI states should follow the data and not the field. Therefore, if the field is a value-type then the field itself should be a custom object. In contrast, if the field is a reference-type then the referenced object should be a custom object. Figure 3 illustrate a field with a value-type where the type of the field has changed from "int" to the custom object named "MESIValueField<int>". The MESIValueField contains all control code to correctly handle the MESI protocol.
Fejl! Henvisningskilde ikke fundet. Figure 4 shows the case of a shared field with a reference-type to an object of type "MyObj" which again contains a shared field of value-type "int". The type of the shared field is now

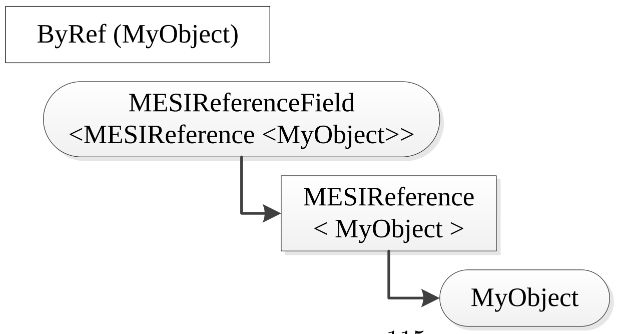

ByRef (MyObject)
MESIReferenceField <MESIReference <MyObject>>
MESIReference < MyObject >
MyObject

Figure 4. A shared "ByReference" field incapsulated in a MESI object.

changed from "MyObj" to "MESIReference-Field<MESIReference< MyObj>>". MESIReference-Field does not contain the MESI protocol; however, it contains functionality to notify others if the field is assigned a new object (reference). The MESI protocol is implemented in the custom object named MESIReference which contains a reference to the actual "MyObj" object.

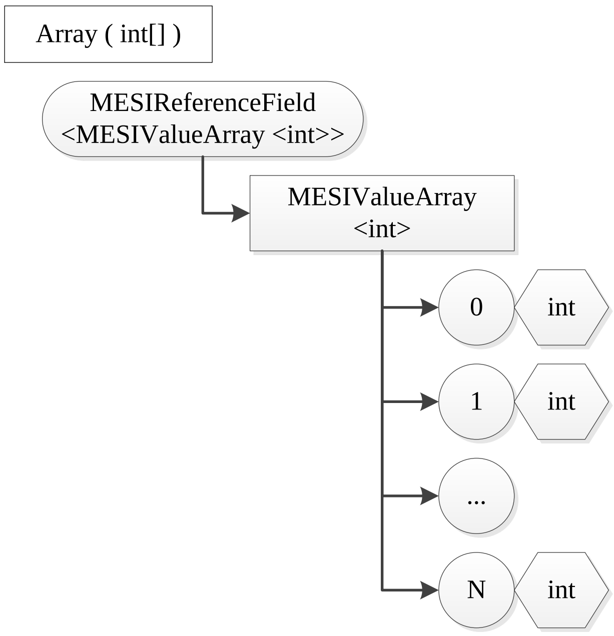


Figure 5. A shared Array field incapsulated in a MESI object.

It should be noted that a special case arise with reference-type fields if the referenced object is of the type Array. The difference is that the real data of the arrays are the elements of the arrays and these are accessed through the CIL instructions Ldelem/Stelem. Therefore, we need a special case to handle arrays which we define as "MESIValueArray" (see Figure 5). This object has a MESI state for each of the elements in the array, but furthermore has support for defining a block size. Thus, enabling the control code to handle blocks of elements in order to minimize the overhead when accessing a large part of an array iterative.

3.3. Code generator

The third component of DistVES is the code generator, which has the responsibility for transforming the original code into a distributed version of the same code. To do this, the code generator must first make a complete tree-based structure of the code to ensure efficient rewriting. The tree contains information on relations between classes and instructions, e.g., the Add instruction takes two arguments, which means that the Add instruction must have two incoming instructions. If the result of the addition is afterwards stored in a variable the Add instruction has an outgoing instruction, which is the store instruction. Furthermore, it is necessary to identify instructions that call another method.

During the actual code generation all Parallel.For loops are identified and the name of the delegate class, which is the body of the loop, is noted. The next step is to modify these classes. As we know that the fields are the only type that can be shared between threads, the fields are a good starting point in order to keep the modifications of instructions to a minimum. As there only are to CIL instructions to access a field namely the Ldfld and Stfld instructions, the code generator looks for these two instructions and when finding them, a recursive modification using the incoming and outgoing instructions starts.

4. Benchmarks

In order to test the performance of the implementation three algorithms were implemented using the `Parallel.For` constructs. The tests were executed on four machines each with an Intel i7-860 processor at 2.8 GHz and 8 GB of RAM. The machines were connected using a Gigabit network through a single switch. The experiments were performed with 1-16 workers (1-4 workers per machine) and repeated five times to get consistent measurements. The tests labeled *Microsoft* `Parallel.For` and `DistVES Parallel.For` was executed on a single machines. Tests labeled *DistVES Network (3)* indicates that one machine ran the server and main client, and the three others machines ran the workers (3, 6, 9 and 12 workers in total). Finally the tests labeled *DistVES Network (3 + 1)* means that all machines ran the same amount of workers giving a total of 4, 8, 12 and 16 workers. In addition one of the machines ran the server and the main client.

The three test applications were written in C# and the source code was not changed between running with the Microsoft .NET `Parallel.For` and the `DistVES Parallel.For` other than a switch indicating which `Parallel.For` method to use.

- Black-Scholes: The algorithm gives the price of European style options and is frequently used in the financial world.
- Ising: A Monte Carlo simulation of the ising model which is a mathematical model for simulating magnetism in statistical mechanics.
- Prototein: Simplification of protein folding with only 2 dimensions and folding in angles of 90 degrees.

4.1. Discussion

The Black-Scholes is an embarrassingly parallel problem; it has very little input data and generates only a single double value as output. Therefore, a good speed-up is expected from both Parallel methods. As seen in Figure 6 linear speed up is achieved using one to six workers.

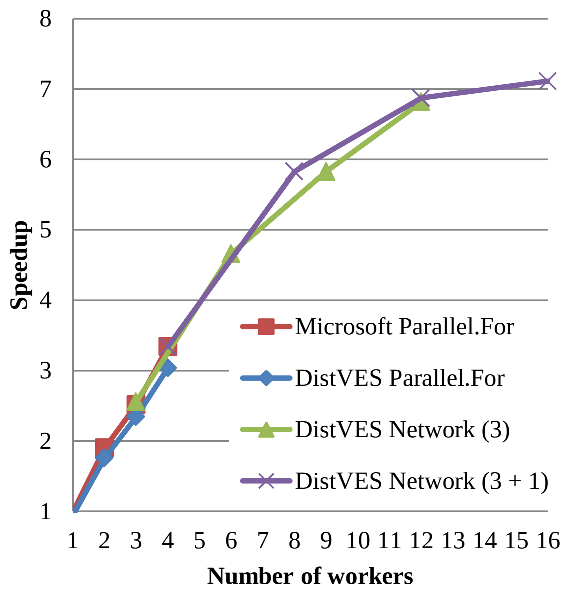


Figure 6. BlackScholes

Afterwards, the two tests using multiple machines still show an increase in speed-up, but with a much lower gradient which flattens as the number of workers increases. The single machine tests show that both methods scales well when running; however, both methods generate some overhead, which result in s scaling that is not perfect. The result is acceptable as the code is much easier to write, than the code required to make perfect scaling. In the network tests, we have a good scaling when using one or two cores per machine; however, using more than three cores result in decreasing utilization. The primary problem is the time span between the main client creating tasks and the initial work being distributed to the workers. The time span was measured to around 25 % of the total running time, when using 16 workers (4 per machine). A secondary problem is the time required in the server to handle messages from clients. The time span is too high between a client sending a request and receiving the responds.

The Ising simulation is Monte Carlo based and thereby embarrassingly parallel as well. On the other hand the Ising simulation contains a barrier to synchronize the calculation of each round. The cost of the synchronization would increase and become the dominating factor if we ran the simulation on a fixed problem size and just increased the number of workers. Therefore, we ran this test increasing the problems size when the number of workers increases. The size of the array for a single worker is 3500x3500 elements, for two workers 7000x3500, for three workers 10500x3500 and so forth. It was not possible to make a run using a total of 16 workers due to memory restraints. When using DistVES these arrays must be transferred even though one of the tests is executed on a single machine, on the contrary Microsoft Parallel use shared memory, and therefore access the memory directly. Therefore, we expect that Microsoft Parallel will scale better than DistVES. A decrease in utilization should furthermore be expected when using the network. As we see in the Gustafsson graph in Figure 7, DistVES is actually outperformed with 20 %, which is a bit high. Nevertheless, none of the four methods are close to the optimal horizontal line. The two network tests show a linear decrease in utilization and when using 12 workers the utilization becomes less than 50 %. The main problem is the overload of the central server and the barriers, but also the high number of accesses to the elements in the array. Each element access has a higher cost in DistVES because of all the bookkeeping required to guarantee consistency among other things.

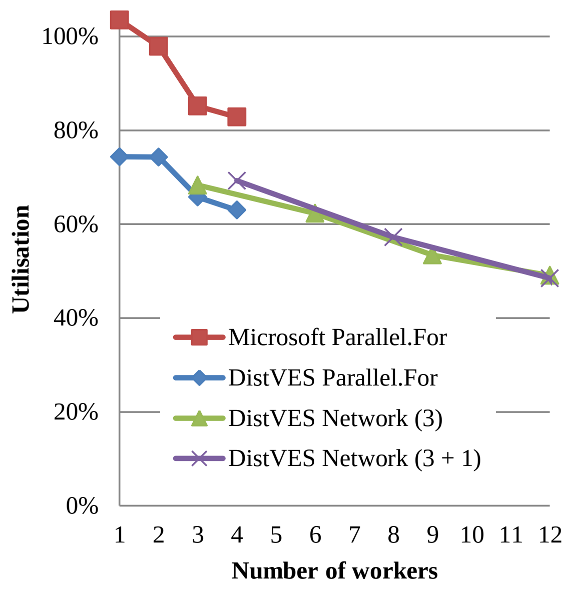


Figure 7. Ising simulation

The final benchmark is the simplified protein folding, which again should yield very good speed-up. The initial step in the program is that the main client creates tasks each containing a partly folded prototein. The tasks are then distributed to the clients, which locally keeps a copy of the fully folded prototein with the highest score. When all prototeins are folded the main client collects the best scores from the workers and finally finds the overall best prototein structure.

As with the Ising simulation, the Prototein folding have an input of some size; however, it is not as large as the Ising simulation. Furthermore, the Prototein folding does not require any barriers to synchronize calculations.

Therefore, the expectations are a linear scaling where Microsoft Parallel will have a better gradient e.g. closer to optimal scaling. As seen in Figure 8 all methods show good scaling, again; however, the two network tests show a decrease when using more than two workers per machine.

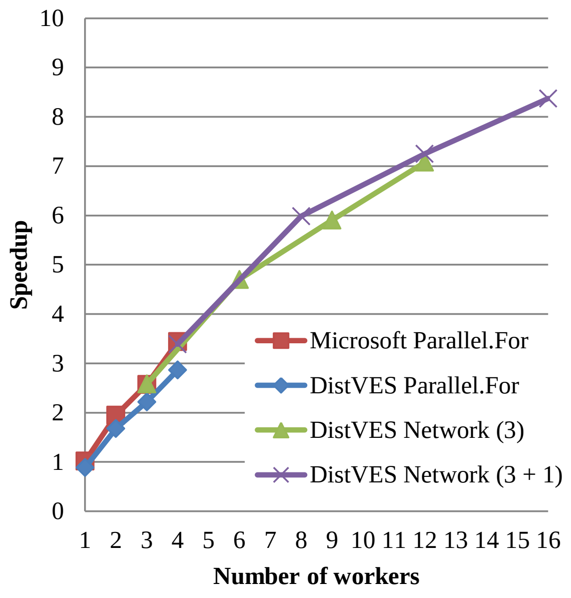


Figure 8. Prototein folding

Furthermore, the two single machine tests show that DistVES scales as well as the method provided by Microsoft; however, DistVES is a bit slower due to more overhead.

The reason for utilization of only 50% when employing a total of 16 workers is the same as seen in the previous tests, namely the cost of the element access.

5. Future work

As seen in the previous section the major problem with the current version of DistVES is the high cost of accesses to shared variables. Furthermore, the consistency model is strict and to gain a better performance a more released consistency model is required. There are a couple of ways to achieved this; either by code annotation, by a more intelligent code generation, or by modifying the virtual machine of .NET (VES/CLR).

It is clear, that code annotation would make it easier to implement a more release consistency model like entry consistency. However, this shifts attention away from making it easier for the programmer to write code, which is essential. Unfortunately, it seems that achieving a good performance is not possible without code annotation.

The second option is to make the code generator more intelligent. A solution could be to categorize shared fields into read-only (write-once) and read-write. Thereby, the control code for the MESI protocol could be skipped by read-only fields and they would work like a normal field, making the performance better.

The best solution is properly to integrate a system like DistVES directly into the VES, but the open-sourced Mono project is the only choice for implementation as the Microsoft implementation of .NET is closed-sourced. The gain is that the user's code does not need to be changed at all and the accesses to shared variable will be the same running with or without a distributed `Parallel.For`. There will; however, be added some overhead when using the distributed version, but it will hopefully be less than in DistVES. The main concern is that the incredible effort it will require to modify the execution engine of Mono will not be justified by the gained speed-up.

6. Conclusion

Improving ease-of-use for scientists with limited programming knowledge to utilize the available hardware on multi-core and cluster computers is very important. Therefore, much effort has been put into making tools that assist the scientists; however, many tools are not widely used and/or will not give the wanted scalability. In this paper we have presented our view on such a system using .NET and a `Parallel.For` construct, which allows the users to easily convert their existing scientific programs into programs that utilize a distributed computer setup. Microsoft has already made support for using the `Parallel.For` construct on a single multi-core machine, but the system described in this paper extent that idea to utilize multiple machines. The implemented test cases show that for some simple scientific problems DistVES scales as well as Microsoft's solution; however, in some cases it does not. Altogether it is should be clear that a parallel programmer's implementation of the tested problems at any time will scale better than the versions using DistVES or Microsofts Parallel Library; however, the two systems can be used by scientists that are not experts in parallel programming and are having a simple scientific application that they want to parallelize. Therefore DistVES cannot be used to parallelize all types of programs, but for a subset e.g. simple scientific application, it will do fine. To improve DistVES a number of ideas, ranging from code annotation to rewriting the VES implementation in Mono in order to support distribution have been proposed as future work.

7. Acknowledgements

The Innovation Consortium supported this research with grant 09-052139.

REFERENCES

[1] A. Geist et al., "MPI-2: Extending the message-passing interface," in *Euro-Par'96 Parallel Processing*, Springer Berlin / Heidelberg, 1996, pp. 128-135.

[2] M. R. B. Kristensen and B. Vinter, "Numerical Python for scalable architectures," in *In Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model (PGAS '10)*, New York, NY, USA, 2010.

[3] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, Aug 1978, pp. 666-677

[4] Microsoft, "Parallel Programming in the .NET Framework," [Online]. Available: http://msdn.microsoft.com/en-us/library/dd460693

[5] OpenMP, "OpenMP," [Online]. Available: http://www.openmp.org

[6] J. Hoeflinger, "Extending OpenMP* to Clusters," Access, 2006, pp. 21-24

[7] T. Seidmann, "Distributed Shared Memory using the .NET framework," in 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2003.

[8] K. Asanovic, "The Landscape of Parallel Computing Research: A view from Berkeley," University of California, Berkeley, 2006.

[9] L. Lamport, "How to Make a Multiprocessor That Correctly Executes Multiprocess Programs," IEEE Transactions on Computers, September 1979, pp. 690-691.

[10] S. V. Adve and H. D. Mark, "Weak ordering - a new definition," In Proceedings of the 17th annual international symposium on Computer Architecture (ISCA '90), 1990, pp. 2-14.

[11] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," In Procedings of the 11th Anual Internation Symposium on Computer Architecture, 1984, pp. 348-354.

[12] H. J.P., "Extending OpenMP* to Clusters," Access, 2006, pp. 21-24.

Running Microsoft .NET code in parallel on a mixed hardware platform

Morten N. Larsen and Brian Vinter*

The Niels Bohr Institute, University of Copenhagen, Blegdamsvej, Copenhagen, DK

Email: Morten N. Larsen - momi@nbi.ku.dk; Brian Vinter*- vinter@nbi.ku.dk;

*Corresponding author

Abstract

For many years, the preferred programming languages for writing algorithms to be executed on large clusters has primarily been C/C++, Fortran. However, but one of the other major programming frameworks, namely Microsoft .NET does not pop up. The reason for this is properly that besides the official tools from Microsoft not many tools exists which can help programmers simplify the process of writing parallel .NET code. Furthermore, most of the official tools only supports a Microsoft Windows [1] or Microsoft Azure [2] platform and not a mixture of platforms including Linux platforms with Mono installed. Finally, some of the most useful tools for writing parallel .NET code only support a single machine, which of course discourage programmers from choosing one of the .NET programming languages for writing parallel program that must scale to multiple machines. In addition, Windows is one of the most commonly used platform used for workstations in companies. Finally, most of these work stations are equipped with multi-core capabilities which are rarely used, it therefore seems natural to employ these machines as part of a cluster. Here, we present the development of a new tool focussing on .NET and based on some of the tools available for simplifying the process of writing parallel code for traditionally programming languages. The tool allows the programmer to call a number of methods which can invoke (send) a method along with parameters and shared data to a central unit. Afterwards the tool distribute the method to a number of connected machines each running one or more workers. By implementing three simple benchmarks, initial tests have shown that good scaling can be archived on a small cluster consisting of Windows machines, and by presenting future design ideas it is believed that it will be possible to scale to must larger mix-platform clusters consisting of internal resources like work stations, server and external cloud resources.

1 Introduction

Simplifying the process of writing algorithms which have the ability to be executed in parallel on large cluster has been the goal of much research over the last couple of decades. Popular tools like MPI [3] and OpenMP [4] are widely used today by programmers wishing to implement algorithms which can be distributed over many computational units. With the blossoming of cloud computing this trend has increased further as it is now possible for people, without resources to buy a large supercomputer, to use cloud instances to make a cheap cluster-computer which they only have to pay for when actual doing computations. The programming languages supported are the traditionally ones such as C/C++, Fortran. which for many years have been the preferred languages among scientists both in terms of speed but especially due to the high amount of well-tested and highly optimized libraries like BLAS and LAPACK. However, Microsoft has during the last couple of years introduced a number of tools like the Task Parallel Library (TPL) [1] and the HPC Pack for .NET [2] in order to simplify the process of writing code which can run in parallel on the .NET platform and the Microsoft Azure Cloud system. The platform supported by the HPC Pack can consists of cloud instances (Azure), servers and work-stations running some sort of Windows, whereas the TPL library currently only supports a single machine and is not directly meant for high performance computing. Thus, there is a gab between using the traditionally programming languages on a mixed platform and using the .NET programming languages on a platform of machines running some sort of Windows. Therefore, we propose a tool which will allow running .NET generated code in parallel on a mixed platform. The goal of the tool is to provide programmers with a set of methods which allows them to run a method in a Fork-Join (see figure 1) style on multiple machines and retrieve the result.

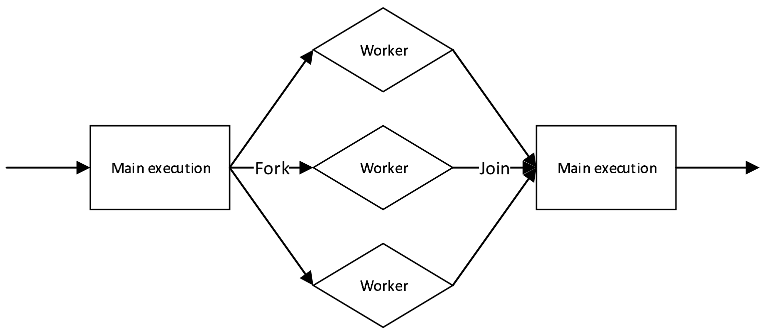


Figure 1: Illustrations of the Fork-Join principle.

The rest of the paper is organised as follows: first a description of the design is given followed by a number of benchmarks. The paper is finalized with a section on proposed future work and finally an conclusion which will summarize the findings.

Related work

As mentioned above Microsoft is the main contributor to research where the .NET platform is the main target. They have two libraries which can be use for parallel computing; one for single machine use and one for a cluster of machines. The TPL library exposes an API with a number of functions for writing both data and task parallel code. A number of loop constructs like Parallel.For, Parallel.Foreach. makes it very simple for programmers to write data parallel code. Programmers can archive task parallelism by using the task construct and making arrays of tasks and then use the invoke method to run the tasks in parallel. Both the data and task parallelism only support a single shared memory machine.

The HPC Pack from Microsoft contains, among other things, a MPI implementation for .NET. [5] It allows running tasks on a cluster made of Windows machines including work stations, servers, and cloud (Azure) instances. Furthermore, it contains a service oriented architecture (SOA) scheduler, a scalable cluster management tools.

Other tools that support .NET exist, including a Distributed Shared Memory (DSM) system which allows for objects to be shared by multiple shared memory machines using a software layer DSM model implemented by using proxy objects. [6] However, the success achievable in scaling this to many machines is unknown, as it is build on top of the executing system (Common Language Runtime) of .NET.

2 Design

This section, will describe the design of the proposed system including defining the API which programmers can use to interact with the system. As described previously, the system is intended to support a mixed platform consisting of work stations, servers, and cloud instances all capable of running the intermediate assembler language of .NET named Common Intermediate Language (CIL) either using Mono or the .NET framework for Windows. This tool is based on the Fork-Join principle by which the programmer can run some code locally, then fork the execution in order to run computational intensive code on a number of machines. When the computational heavy part finishes, the system joins the results on the client side and the code can continue execution locally. The Fork-Join principle is well-known from other such tools like OpenMP, Microsoft TPL.

2.1 The basic design

The TPL tool makes it simple to fully utilize a multi-core machine, as the programmer can just use a Parallel.For or define a number of tasks which may be executed in parallel. However, even though it is simple for the programmer to use, the tool does not encourage the programmer to consider how data is shared between tasks and therefore the risk of writing code containing race conditions is high. in contrast, the advantages of this approach is that it makes the scheduling easier for the TPL as there is no relation between tasks and that the TPL does not need to include a model for ensuring consistency of any shared data. A previous attempt [7] to implement a consistency model in .NET, turned out to be difficult as no information can be derived from the CIL code. In addition, the fine grained consistency on top of the .NET execution system (VES) is very expensive when the programmer does not provide extra information regarding how a program accesses shared data. This design along with the use of an atomic operation to access/change the loop variable of any Parallel loop constructs, are some of the main reasons why TPL only support a single machine. Therefore, another approach is needed and preferable one where programmers are strongly encouraged to consider how they access data and at the same time implicitly provide this information to the tool. A well-known paradigm which does this is Communicating Sequential Processes (CSP) defined by C.A.R. Hoare [8] in the late seventies. CSP defines a program as a number of small processes connected to each other using directed channels. Shared data between two processes exists only in the sense that one processes can send data to another using the channel between them. This way of sharing data is very different from the shared memory approach, but it forces the programmer to consider how data is accessed. In addition, it is easier for the programmer to eliminate race conditions and dead locks when each process has a size (tiny) which makes it possible to verify that no errors exist. Moreover, CSP contains methods to formally prove algorithms written using CSP; however, as this will not be used in this tool it will not be further discussed. Inspecting the practical part of CSP, a process can be thought of as a black box taking some input and based on this input returns one or more outputs. The input and output are respectively done by reading/writing to channels which is a channel can be though of as a shared piece of data. CSP defines multiple channel-types like One-2-One, Any-2-One, Any-2-Any [9], where *any* means that multiple processes can be attached. If multiple processes are connected into one end of a channel, the system defines numerous ways of deciding which process is given access to the channel like round-robin, random, prioritize. Accessing a channel is in the basic case a blocking operations that will first be executed when the other end of the channel is accessed, meaning that data is first transferred when both writer and reader are ready to communicate. This makes it easier for the programmer to reason about a program as communication deals with both transferring data and acts as a synchronization point between the channels to end-points. So how can this be transferred to .NET without adopting the whole CSP scheme of processes, channels etc., as we do not wish to force the programmer to write CSP programs, but just want to encourage the programmer to consider how she can access shared data. One possible solution is to only allow the programmer to run methods in parallel that do not access shared data and then use input parameters and return values to adopt the reading and writing of channels. This approach does not fully adopt the CSP way of thinking, as a .NET method can only have one return type. Additionally, the changes made to any reference-typed input parameter within the body of the method will also be available outside the method. Finally, the synchronization point defined when accessing CSP channels is non-existing in this approach. The first issue can luckily be solved in many ways e.g. by making a special return type that could encapsulate multiple values like a tuple. The second problem can only be solved by not allowing the programmer to change input parameters unless the parameters is also returned as the return value of the method. However, controlling that the programmer does not violate this constrain is difficult as it requires a total recursive traversal of all accesses made to the parameters as the parameter (an object) can be loaded and methods can be called on the object which will modify it. The third problem of not having a well defined synchronization point is in reality not a direct problem in most cases, and if a hard synchronization point is needed it can be solved by dividing the method into two methods. Then the programmer can invoke the first method and use the result as input to the invocation of the second method like illustrated below:

```
result1 = Invoke(MethodOne, Paramaters);
// Implicit synchronization point
result2 = Invoke(MethedTwo, result1);
```

The problem of accessing shared data, deals with how changes made to data is distributed among all machines using the data, especially if the data access is fine grained like accessing each element in arrays. However, if no changes are made to a shared data item, there is no need of special handling of the accesses. As the intended use of this tool will lead to the same method being executed multiple times with different input parameters, it seems sensible to allow read-only access of shared data namely Fields in .NET jargon. As with parameters it is hard to detect if the programmer violates this constrain; however, as it is less intuitive that one cannot write to shared data (fields), we define that shared data items must be of the value-type or multi-dimensional arrays containing value-typed elements which can easily be verified by ensuring that all fields is of the mentioned types and are only accessed using the "load field" CIL instruction named `ldfld`.

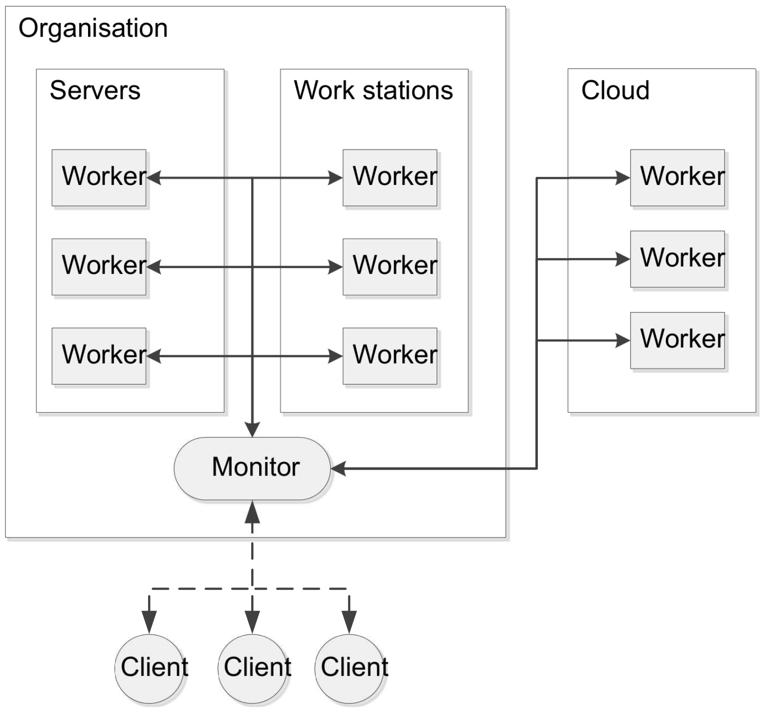


Figure 2: Overview of the components defining the tool.

The last step is to consider the return type of a method, especially the case where the programmer wishes to use it as an input parameter invoking another method. If this is the case it is preferable that the data stays on the server-side of the application, and that it is not transferred back and forth between the client-side and the server-side. This will furthermore, in combination with asynchronous invoked methods, enable the client to disconnect from the server side and return later to check if the calculation has finished and eventually retrieve the result.

This finalizes the basic idea behind the tool and some of the initial design decisions and the next step is to define the component of the system, how components are connected and finally the API.

So far two components have be mentioned; namely, a client and server-side component. The server-side will furthermore consists of two components; a monitor and a worker component (see figure 2). The programmer will use the client to send jobs to the monitor which will distribute the job (divided into tasks) to the connected machines each running one or more workers.

The monitor will be responsable for transforming a job consisting of a method, some parameters and possibly some shared fields into tasks which the workers can execute. The first step is to control that the requirements to the method mentioned above are not violated and determining if any shared fields are read.

If this is the case the monitor must request the values (data) of the accessed fields from the client before it can proceed. After the monitor has received all the data, it will create a ResultToken which the client can later use to access the result of the invocation.

Before the client invoke the system it will have created a class (instantiated object) to which the method to be invoked belongs to. This class possibly contains other methods which perhaps are to be invoked later, shared fields, and the parameters which are going to be used to execute the invoked method. However, as methods in .NET needs a target (instantiated object) unless they are static, all workers need the instantiated object mention above. Unfortunately, due to the fact that the workers does not share memory, the monitor must transfers the instantiated object to all workers which requires that it is serialized. In order for a class to be serialized in the first place, it must have the *Serializable* attribute. This and the fact the the only code really needed by the workers is the method to be executed another approach is chosen. This approach bases on defining a basic class which at runtime can be modified to include the method to be executed along with the fields which are accessed. The constructor of this class named *Executor* takes as argument; a list of values which the constructor uses to initiate the shared fields. The modified *Executor* class can now be distributed to all workers and they can instantiate the class which will make it possible to call the "parallel" method.

As the parallelization of method lies in the parameters meaning that the same method is called many times using different parameters, a methodology is needed to define these sets of parameters. It was therefore decided to define the parameters using a list of values for each parameter along with information about how the monitor should mix these lists together to make a single list of parameter-sets. The three ways of adding a new parameter are defined as "element wise", "single element", and "zip". These are used to define how the "new" parameter is to the already constructed list of parameter-sets. Initially, the list is empty and the first list of parameter values are added, then the next list of parameter values are added and so on. Examples of the resulting parameter-sets for the addition of the parameter list (a,b,c) to the existing parameter list (1,2,3) are shown below:

```
Element wise: ([1,a],[1,b],[1,c],[2,a],[2,b],[2,c],[3,a],[3,b],[3,c])
Single element: ([1,(a,b,c)],[2,(a,b,c)],[3,(a,b,c)])
Zip: (requires same length): ([1,a],[2,b],[3,c])
```

As each method will use a single parameter-set as input and generate a single output (result), the outcome of using all parameter-sets is a single list of outputs. Each worker could send the result back to the monitor, which would later return the results to the client, but if the results were to be used as input for another method this is a waste of resources. An alternative approach is to let the workers keep the results locally and only send them to the monitor if requested by the program. This approach will minimize the amount of data transfers, but naturally increase the waiting time when the final results is requested. In addition, this approach requires that the monitor knows where the results are located. This problem will be touched upon below when discussing the scheduler.

The next step is to define the API. Firstly, a method is needed to invoke the tool. An invocation will return a ResultToken, which can be used to fetch the result from the monitor:

```
class Client
{
    ResultToken = Invoke(ID, Method, List of selector, Parameters, List of ResultTokens)
}

class ResultToken
{
    bool = IsDone()
    List = GetResult()
    Delete()
}
```

In the implementation, generics should be used on both the ResultToken and definitely on the Invoke method to ensure that the types of the parameters and/or ResultTokens used as input in combination with the selectors matches the types of the input parameters of the method which the user wishes to invoke. Likewise, the output type of the method should match the type of the resulting ResultToken. This will help the programmer not making mistakes.

The performance in a system like the proposed depends strongly on how well the monitor will distribute tasks to the connected workers. Many methods exists ranging from a simple centralized round-robin distribution to distributed and auto-balanced algorithms like work-stealing. As it has already been established, workers will not transfers results back to the monitor every time they finish a task, a simple round-robin distribution of tasks will naturally result in an increase of data transfers if the tasks being distributed use results from previously tasks as input. In such situations it is preferable if the monitor is capable of sending a given tasks to the worker which holds the results needed in order for the task to execute. For the monitor to be able to do this, two pieces of information are needed; namely which worker executes a given task and the size (in bytes) of any given result. One could argue that both pieces of information in most cases would be present at the monitor-side when it schedules the tasks because the monitor knows which worker will receive a given task and if the result of execution is one of the primitive types in .NET (int, float, double etc.), the size of the result is also known. However, if the return type is an object or an array, the size will be unknown at the time of scheduling. Therefore, and as the overhead of sending a single packet for each executed tasks is small, it was decided that the worker will report back which tasks they have solved including the size of the result. Now the monitor will be able to use the information to schedule the tasks based on where the input parameters to a method are located. Instead of doing this scheduling centrally an alternative solution is to let the workers request tasks from the monitor and then let the workers inspect the input information. If most of the data is remotely located, the worker could forward the tasks to the worker having most of the data. This will partly distribute the scheduling, but at the risk of the scheduling becoming unbalanced. The "Future work" section of this paper will discuss a possible solution for solving this. As there will be a small overhead for the monitor to process each task-request from a worker, the workers should be able to request tasks asynchronously, so that the overhead of communication and scheduling would be mostly hidden by calculations.

3 Experimental

To test the implementation of the design, three simple benchmarks were implemented. The benchmarks were executed on a small cluster consisting of four machines connected through a gigabit network. Each machine was equipped with a quad-core Intel i7 processor with HyperThreading running at 2.8 GHz and a minimum of 8 GB of memory at 1.333 MHz. The execution of each benchmarks was repeated four times using one to sixteen workers. The workers were distributed among the machines in two ways, namely by round-robin or by filling one machine (max four workers per machine) at a time. The three benchmarks chosen was a prototein folding, Black-Scholes and a K-nearest-neighbors (kNN).

3.1 Prototein

Prototein folding [10] is a simplified two dimensional version of the well-known Protein folding, using only two amino-acids (H and P). Furthermore, the amino-acids are limited to only having four neighbors in 90 degree angles to one another. Given a string of amino-acids the goal of the algorithm is to minimize the number of H amino-acids in the string not having four neighbors. The algorithm is bag-of-task parallel and therefore it is expected to scale linearly.

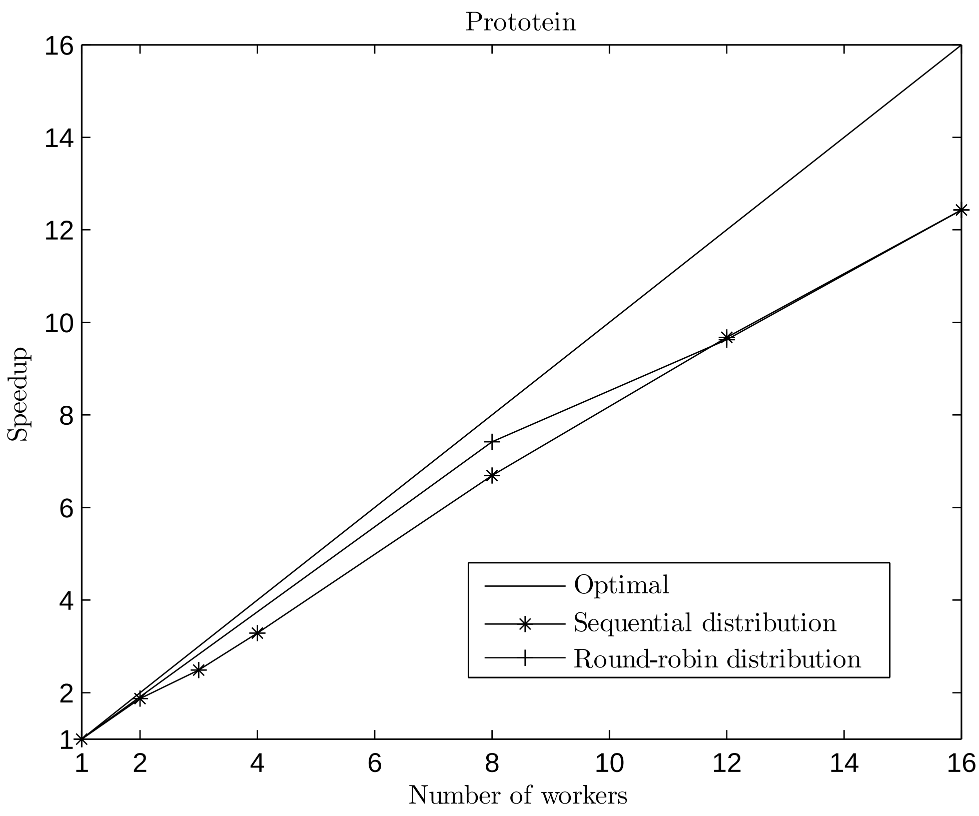


Figure 3: The result of running the Prototein benchmark.

Figure 3 illustrates the result of running the benchmark. Starting with the test in which the workers were distributed by filling one machine at a time, it can be seen that linear scaling is achieved and when using one and two workers the gradient is close to the optimal. As all four cores shares the same memory bus, the the bus can not get data fast enough from the memory to the computational cores and we expect this to be the reason for the decrease of the gradient when using more than two workers per machine. When distribution workers using the round-robin model, the same is seen, but as workers are equally distributed between machines, the gradient is closer to the optimal until eight workers (two per machine), and then decreases. Thus, if one uses more than two workers per machine the additional workers (cores) will only contribute with approximately half of their potential.

3.2 Black-Scholes

The Black-Scholes is a Monte Carlo simulation used for pricing European-styled bonds and it is embarrassingly parallel and should scale close to linearly. The result of the execution is shown in figure 4 and when comparing the result to the one given in figure 3, one can see that the two graphs are almost identical, so again the problem scales linearly, but with a lower gradient when using more than two workers per machine.

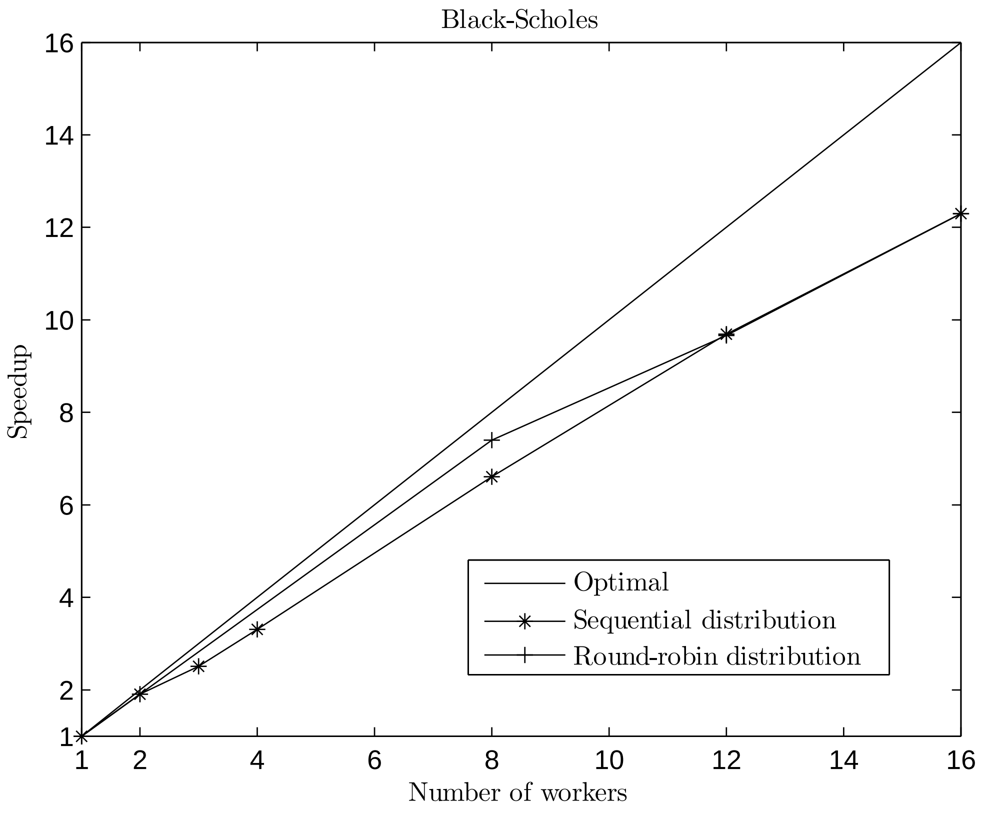


Figure 4: The result of running the Blacks-Scholes benchmark.

3.3 kNN

The final benchmark is the well-known k-nearest-neighbors, kNN problem. Given a number of datum in a N-dimensioned space, the goal of the algorithm is to find the k nearest (by distance) neighbors for all datum. The algorithm is very simple, but it requires a large amount of datum meaning that more data must be transferred. Furthermore, given a single particle the calculational needed in order to calculate the distances to all other datum is very small, and therefore each task should calculate the distances between multiple datum in order to increase the calculation/commonunication ratio. With a sensible task size the algorithm is expected to scale linearly possibly with a lower gradient compared to the two previous benchmarks due to increased amount of data transferred.

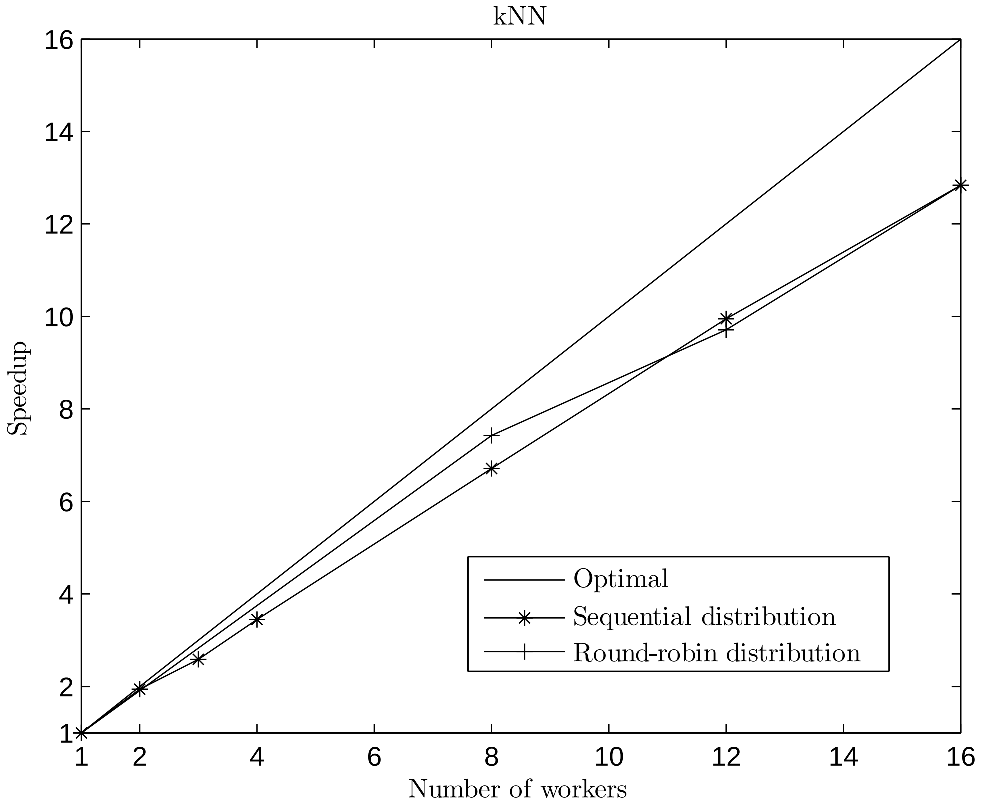


Figure 5: The result of running the kNN benchmark.

Figure 5 shows, as the figures of the previously two benchmarks, that once again the problem become memory bound. However, the gradient is the same as the gradient seen in the graphs for the previous two benchmarks, which means that the data transfers are hidden due to doing overlapped execution.

4 Future work

As the proposed scheduling method can lead to an unbalanced distribution, a more advanced method for distributing tasks is needed. One possibility is to use work-stealing where each worker has a double-ended queue with tasks prioritized based on the amount of data that is locally present. Alternatively, a more traditionally approach could be used where the monitor do the initial distribution by prioritizing the tasks and then let the worker use a work-stealing implementation based on a normal double-ended queue.

Another issue concerns security when using cloud instances. With a platform consistency of both internal and external machines it may be preferable that programmers can mark certain data with a special attribute which specify that the data must not be transferred to the external cloud instances. Likewise, the programmer should also be able to specify if jobs or return values should be exclusively be shared among the internal machines. In order to be able to do this, the monitor must be able to distinguish between workers running on the internal and external machines. This could be done by implementing a management module, allowing for starting/shutdown cloud instances, servers, and work stations.

5 Conclusion

This paper presents a simple tool which allows code written using one of the Microsoft .NET programming languages to be executed in parallel on a mixed platform consisting of machines running Windows and/or Linux. The tool exposes a small API along with some requirements to the method which the programmer wishes to execute in parallel. The tool analyses and modifies the code, before the monitor converts the provided parameters into a number of tasks, which are later distributed to the connected workers. Initial tests indicates that for the benchmarks linear scaling can be achieved, but all benchmarks show that if using more than two workers per machine the gradient of the scaling decrease due to hitting the memory wall.

Acknowledgements

The authors would like to thank the The Innovation Consortium for supporting this research with grant 09-052139.

References

1. Microsoft: **Parallel Programming in the .NET Framework**. *http://msdn.microsoft.com/en-us/library/dd460693*. [Accessed march 2013].
2. Microsoft: **Microsoft High Performance Computing for Developers**. *http://msdn.microsoft.com/en-us/library/ff976568.aspx*. [Accessed march 2013].
3. Geist A, Gropp W, Huss-Lederman S, Lumsdaine A, Lusk E, Saphir W, Skjellum T, Snir M: **MPI-2: Extending the message-passing interface**. In *Euro-Par'96 Parallel Processing*, *Volume 1123 of* Lecture Notes in Computer Science. Edited by Bouge L, Fraigniaud P, Mignotte A, Robert Y, Springer Berlin Heidelberg 1996:128–135, [http://dx.doi.org/10.1007/3-540-61626-8_16].
4. OpenMP: **OpenMP**. *http://openmp.org*. [Accessed march 2013].
5. Microsoft: **Microsoft MPI**. *http://msdn.microsoft.com/en-us/library/bb524831(v=vs.85).aspx*. [Accessed march 2013].
6. Seidmann T: **Distributed shared memory using the .NET framework**. In *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on* 2003:457 – 462.
7. Larsen M, Vinter B: **A Distributed Virtual Machine for Microsoft .NET**. *Journal of Software Engineering and Applications* 2012, **5**(12):1023–1030.
8. Hoare CAR: **Communicating sequential processes**. *Commun. ACM* 1978, **21**(8):666–677.
9. Vinter B, Bjørndalen JM, Friborg RM: **PyCSP Revisited**. In *CPA*, *Volume 67 of* Concurrent Systems Engineering Series. Edited by Welch PH, Roebbers HW, Broenink JF, Barnes FRM, Ritson CG, Sampson AT, Stiles GS, Vinter B, IOS Press 2009:263–276.
10. Hayes B: **Computing Science: Prototeins**. *American Scientist* 1998, :216–221.