

A Domain Specific Meta Language to Abstract Concurrency Concerns in Multicore Computers

Jakob Ehmsen
Esbjerg Institute of Technology - Aalborg Universitet,
CIS4 Spring 2009, Distributed Systems,
je1536@student.aau.dk,
Supervisor Daniel Ortiz-Arroyo

6. July 2009

Summary

The performance of traditional computers which contain a single processor is stagnating. The current trend of several chip makers is now to release chip multiprocessors (CMPs). Manycore chips, including 1000 core chips, are realistic to emerge in the relatively near future. This trend and foresight implicate mandatory use of parallel computing within software development - i.e. if sustaining the performance improvements achieved so far is the objective. Currently, only few software developers adopt parallel computing in practice. Hence, a transition from sequential programming to parallel programming is necessary for many software developers. Implicitly parallel programming models have been advocated as a solution for that transition.

An implicitly parallel graph meta language (GML) was designed, constructed, and evaluated. GML is compiled into a higher level language and is therefore a meta language. The target language is Erlang. The target code consists of an application of an intermediate graph library (IGL). IGL and GML was constructed in relation to a graph application algorithm. The parallelization of GML code can be summarized as follows. First, identify concurrency inherent in the GML code. Second, using the identified concurrency, extract a data flow graph consisting of threads, forks, joins, and/or parallel loops. Third, generate the target code according to the constructed data flow graph. The theoretical improvements of an implementation of the graph application algorithm in GML were significant compared to a sequential implementation.

For the purpose of testing GML, a benchmark was designed and constructed. The benchmark consisted of three implementations of the graph application algorithm: a sequential implementation, an explicitly parallel implementation (EPI) in Erlang using IGL, and finally multiple implicitly parallel implementations (IPIs) using GML. In addition, the implementations were run on three different platforms: a duo core platform (DCP), a quad core platform (QCP), and an SMT platform (SMTP). The main success criteria of the benchmark was maintenance of historical performance improvements. The benchmark also concerned speedup, efficiency, and algorithmic- and architectural scalability.

For SMTP, the IPIs were unsuccessful. Algorithmic scalability and maintenance of historical performance improvements were not achieved. In addition, the theoretical improvements were not precise.

On the other hand, the implementations for the CMP-based platforms (DCP and QCP) showed success. The speedups and efficiencies of the IPIs were more or less equivalent to those of EPI. Theoretical improvements were somewhat precise. Algorithmic- and architectural scalability were achieved. Finally, and most importantly, the IPIs achieved maintenance of historical performance improvements, especially for QCP.

Contents

1	Introduction	4
2	Main Issues within Parallel Processing	7
2.1	Parallelism VS Concurrency	7
2.2	Granularity of Parallelism	8
2.3	Identification of Concurrency	8
2.4	Computer Architecture	11
2.5	OS Support	13
2.6	Parallel Programming	13
2.7	Parallelizing Compilers	14
2.8	Performance	14
2.9	Scalability	15
3	Parallel Processing Programming Models	16
3.1	Shared Memory	16
3.2	Message Passing	17
3.3	Relation to Computer Architecture	19
4	Compiler Theory	20
4.1	Scanner	20
4.2	Parser	21
4.3	Semantic Analyzer	22
4.4	Source Code Optimizer	24
4.5	Code Generator and Target Code Optimizer	24
4.6	Compiler Construction Tools	24
5	Related Work	26
6	The Erlang Programming Language	28
7	Graphs	31
7.1	Fundamentals	31
7.2	Algorithms	31
7.3	Application	32
8	Graph Library and Meta Language	33
8.1	A More Explicit Algorithm 1	33
8.2	Graph APIs	34
8.3	Meta Language	35

8.3.1	Grammar	35
8.3.2	Parallelization	36
8.3.3	Target Code Generation	40
8.3.4	Theoretical Improvements for Parallelization	42
8.3.5	Correctness	43
8.3.6	Limitations	44
9	Benchmark	45
10	Evaluation	48
10.1	DCP	48
10.1.1	G_c	49
10.1.2	G_s	50
10.1.3	Comparison	51
10.2	QCP	51
10.2.1	G_c	52
10.2.2	G_s	53
10.2.3	Comparison	54
10.3	SMTP	55
10.3.1	G_c	55
10.3.2	G_s	56
10.3.3	Comparison	57
10.4	Overall Comparison	57
11	Conclusion and Future Work	59
12	Acknowledgements	61
A	Extraction of Implicit Information in Algorithm 1	62
B	Mapping Algorithm 2 to IGL APIs	64
C	Benchmark Implementation Outcomes	65
C.1	SI	65
C.2	EPI	66
C.3	IPI outcomes	69

Chapter 1

Introduction

In 1965, Gordon Moore saw a trend in computer hardware: since 1959 the number of transistors placed on chips had increased roughly by a factor of two per year [74]. This observation was later coined as “Moore’s Law” by Carver Mead [102, 33]. Gordon Moore expected that this rate would over the short term continue and perhaps even increase whereas this increase of rate over the longer term was more uncertain [74]. Gordon Moore has later stated, in 2003, that: “...no physical quantity can continue to change exponentially forever.” - “Your job is delaying forever.” [75].

Today, Moore’ law still holds water [33]. Unfortunately, the performance of traditional computers which contain a single processor is stagnating due to various problems, such as heating and power consumption issues [64]. Thus, the use of the additional transistors must change. The conventional wisdom is now to double the number of cores on chips for each silicon generation [13] (multi-core architectures and parallel computer architectures in general are presented and discussed in Section 2.4). Consistent with this wisdom, the current trend of several chip makers, e.g. Intel®, IBM®, AMD, and Sun™, is to release multicore architectures [7].

In the future, many-core architectures, say 16, 32, and above cores, are likely to become ubiquitous. Currently, there are many discussions concerning the future multicore processors from Intel® and AMD, such as 6- [78, 79], 8- [80], 12- [78], 16- [79], 32- [76], 64- [77], and 80-core processors [81]. Actually, Intel® has demonstrated a 64-core architecture [94]. Based on Moore’s Law, then 1000 core chips will emerge within a decade [4]. Furthermore, there exists evidence that when 30nm technology becomes available it is possible to achieve these 1000 core chips [13]. Since Intel® has already demonstrated a 32nm chip and they plan to begin production of 32nm microprocessors in 2009 [55, 56], 1000 core chips seems to be just around the corner.

Clearly, the current and possible future development within hardware implicates a need for parallel computing within software¹. Parallel computing consists of applying multiple processors (or a parallel computer) to solve a single computational problem [10, 61, 72, 52, 50]. The term parallel processing shares this definition in much literature, e.g. in [36], [2], [108], [10], and [60]. Therefore,

¹If not already applied.

parallel computing and parallel processing are used interchangeably from now. A parallel computer is simply a computer system with multiple processors which support parallel programming [89] (see Section 2.4 for more details). Parallel programming consists of splitting a single computational problem into parts and subsequently distributing execution of these parts on separate processors [107].

For the existing sequential software and future software, parallel computing is essential in order to sustain the yearly performance improvements achieved so far. So, adaptations within software are most likely needed². The simplest adaptation to parallel computing for a software developer, would be to embed parallel processing on the hardware level, i.e. running multiple instructions simultaneously. This would yield a solution where software developers could continue developing software with their current repertoire. Alas, there is only little exploitable parallelism on the instruction level and the shift must be handled on the thread level [64]. The “free performance lunch”³ is over for us programmers [98]. James Reinders a senior engineer and the director of business development and marketing for Intel®’s Software Development Products even predicts that “Within a decade, a programmer who does not think “parallel” first will not be a programmer.” [90].

Though the importance of parallel programming is clear, “only graduate students and other strange people write parallel software.” and “...professional software engineers almost never write parallel software.” [71]. Thus, a software paradigm shift within mainstream software development seems to be necessary - moving from sequential programming to parallel programming.

Before parallel programming can be performed by a developer, parallel programming must be learned. Much theoretical material about this subject is available⁴. On the practical aspect, a programming language must be used at some level. A programming language has some level of abstraction (machine, assembly, high-level, and so on). Introducing a developer to parallel programming with a level of abstraction too low might result in the developer focusing on non-concurrency oriented constructs, such as registers, jumps, and interrupts at the assembly level. Raising the level of abstraction to a language containing primitives for concurrency, such as Erlang (detailed in Chapter 6), would probably be more effective from a learning perspective. With respect to performance of applications, results have shown indications that the level of abstraction should be even higher [35]. Implicitly parallel programming languages might provide an appropriate abstraction level. With such a language, a software developer would probably focus more on the business logic than the parallelization of an application.

In addition to the above, implicitly parallel programming models has been advocated for programming against manycore processors in [54]. Similar to the indications of the results in [35], they argue that explicitly parallel programming is likely to be counterproductive for most programmers over the longer term. Some of the issues presented related to explicit parallelization are: determining

²Again, if not already applied.

³The free performance lunch can be summarized humorously by quoting Joel Spolsky [97]: “As a programmer, thanks to plummeting memory prices, and CPU speeds doubling every year, you had a choice. You could spend six months rewriting your inner loops in Assembler, or take six months off to play drums in a rock and roll band, and in either case, your program would run faster. Assembler programmers don’t have groupies.”

⁴For instance, the material in Chapter 2 and the literature referenced in that chapter.

granularity of parallel execution, setting up data structures which allow correct parallel execution, porting issues, i.e. repeating the experimentation process for successive processors, and difficulties regarding composition of explicitly parallel code. They argue that implicitly parallel programming models in conjunction with appropriate compile tools and hardware will be a better solution.

Due to the above, a language is designed, constructed, and evaluated. Designing and constructing a full-fledged general purpose implicitly parallel programming language is unrealistic because limited time is available. A domain specific language from which future generalizations might be drawn is selected as a replacement. For applicability of the language, the domain of graphs is selected (an application is exemplified in Section 7.3). This is a rather large domain and only a limited support of this domain is part of the language to be constructed. Based on the author's experience of machine- and assembly language, compiling into these languages is too large a task. A different approach is selected: the source language is compiled into a higher level language resulting the source language into being a meta language. Further, the target language selected supports explicit expression of concurrency and is one which the author is familiar with. More specifically, Erlang is selected as the target language. Compiling an implicitly parallel language to Erlang is probably a complicated affair. In addition, the author has limited experience within compilation. It is therefore decided that the source code written in the implicitly parallel graph meta language (GML) is compiled into an application of an intermediate graph library (IGL) written in Erlang. In addition, for relevance and simplicity, multicore architecture retrieves the main focus.

To evaluate GML, a software developer new to parallel programming is required. Though the author has gained some experience within parallel programming, this experience can still be considered to be limited. The author is therefore selected for the evaluation. In addition, GML should be compared to an explicitly parallel programming language. For this, Erlang is selected.

To realize the design, construction, and evaluation of GML, a number of specializations are made later due to dependencies on prerequisites. For the evaluation, a number of specific evaluation measures and a graph application are selected. The graph application will serve as a mean to derive the subset of the graph domain, the IGL APIs, and the primitives of GML.

Following the approach mentioned above, the report is structured as follows. The reader is first introduced to main issues within parallel processing in Chapter 2. After this, the dominant parallel processing programming models are presented in Chapter 3 followed by a summary of compilation theory in Chapter 4. Next, in Chapter 5, some of the work related to this project is summarized. After this, the Erlang programming language is presented in Chapter 6. Following this, the fundamentals behind graph theory, graph algorithms, and a graph application are covered in Chapter 7. Then, IGL and GML are introduced in Chapter 8. In Chapter 9, the benchmark for the evaluation is covered followed by Chapter 10 containing the evaluation of the GML applications and the native Erlang graph applications. Finally, Chapter 11 concludes this work including possible future directions.

Chapter 2

Main Issues within Parallel Processing

Parallel processing is a domain containing several issues. This chapter starts by clarifying the terms parallelism and concurrency. Next, granularity of parallelism is defined and discussed shortly. Following this, the issues within parallel processing important for this project are covered, which are identification of concurrency, computer architecture, OS support, parallel programming, parallelizing compilers, performance, and scalability.

2.1 Parallelism VS Concurrency

Having a common understanding of the terms parallelism and concurrency is necessary for a coherent discussion of parallel processing. In this section, a clarification of these terms is made.

According to [100], there is no agreement on the definition of concurrency and how concurrency relates to parallelism. They present the following three different conventions commonly used:

- In the first convention, it is possible to have concurrency in a programming language without parallel computers whereas parallel execution can occur without concurrency in a programming language. Put in another way, concurrency is potential parallelism.
- In the second convention, concurrent is used to describe simultaneously executing processes which may interact with each other and parallel to describe simultaneously executing processes which are independent of each other.
- In the final convention, parallel and concurrent are synonyms.

Throughout this report, the first convention is adopted.

2.2 Granularity of Parallelism

The granularity of parallelism is an expression of the number of computations performed in parallel between synchronizations [32]. The more coarse grained the parallelism is the greater this number of computations is. The more fine grained the parallelism is the less this number of computations is.

Whether to choose fine-grain or coarse-grain parallelism is probably context-dependent. Intuitively, the greater the number of processors is the more fine-grained the parallelism should be. Some results have indicated that loop-iteration level parallelism performs better for a limited number of processors [24].

2.3 Identification of Concurrency

In order to realize any parallel software, the identification of concurrency in computational problems is essential. In general, concurrency can be found where no dependencies among computational tasks exist. Concurrency can also be found by breaking dependencies among tasks. Therefore, identifying both dependent and independent tasks are essential skills within parallel programming. The ideal parallel computations to obtain are called embarrassingly parallel computations [107]. Such computations consist of sequences of tasks where these sequences are independent of each other.

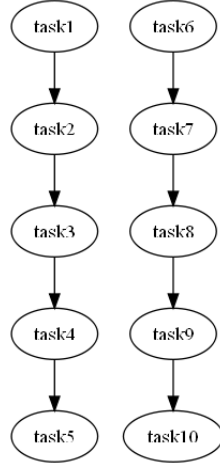
One way to identify dependencies and independencies among tasks is by constructing a data dependence graph (DDG) [89]. A DDG is a directed graph (see Section 7.1 for more details concerning graphs) where each vertex represents a task which needs to be finished. An edge from vertex x to vertex y in such a graph means that task x must finish before task y - or put differently, task y is dependent on task x . If no path exists between x and y , then x and y are independent tasks. Hence, these tasks may be executed in parallel.

Additionally, in a DDG, one can refer to three fundamental kinds of task patterns. The first pattern is purely sequential dependence [89], which corresponds to representing sequential computations. This pattern is illustrated in Figure 2.1. Embarrassingly parallel computations can be perceived as computations consisting of sequences of purely sequential dependent tasks. This is depicted in Figure 2.2.

Figure 2.1: Illustration of a purely sequential dependency. *task2* is dependent on *task1* and *task3* is dependent on *task2*.

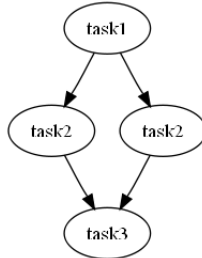


Figure 2.2: Instance of an embarrassingly parallel problem. The sequences (*task1*, *task2*, *task3*, *task4*, *task5*) and (*task6*, *task7*, *task8*, *task9*, *task10*) are independent of eachother.



The next pattern is data parallelism [89], which occurs in situations where multiple independent tasks perform identical operations to distinct elements of a data set. Data parallelism is illustrated in Figure 2.3. This kind of dependency pattern often appears in sequential code in the form of loops. An example of such code is provided in Listing 2.1.

Figure 2.3: Illustration of data parallelism. The two identical *task2* operations are dependent on *task1* but are independent of eachother and may therefore be executed in parallel. *task3* is dependent on both of the *task2* operations.



Listing 2.1: Code example of data parallelism. Each storage location in the array *a* is assigned to 0. In a sequential manner, this kind of operation is performed *n* times. In a parallel manner, all these operations can potentially be performed in parallel on an *n*-processor machine.

```

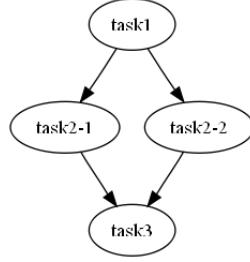
for i := 0 to n
    a[i] := 0

```

The final pattern is functional parallelism [89] (also referred to as task-level concurrency, for instance in [91]). This is concerned with situations where multiple independent tasks perform distinct operations on distinct data elements.

This kind of pattern is illustrated in Figure 2.4. A code example is also provided in Listing 2.2.

Figure 2.4: Illustration of functional parallelism. The two tasks *task2-1* and *task2-2* perform distinct operations and are dependent on *task1* while they are independent of each other and can be performed in parallel. *task3* is dependent on both *task2-1* and *task2-2*.



Listing 2.2: Code example of functional parallelism. The two statements each perform a distinct operation. In the first statement, the variable *x* is assigned to the addition of *a* and *b*. In the second statement, the variable *y* is assigned to the production of *a* and *b*. These two statements are independent and can be performed in parallel.

```
x := a + b
y := a * b
```

Furthermore, there are different kinds of dependence relations between tasks. These can be divided into control dependencies and data dependencies [32].

Control dependencies correspond to computations which are executed as a consequence of the control flow [32]. An example of such a dependency is presented in Listing 2.3.

Listing 2.3: Code example of control dependency. The value of *y* depends the branch target *x > 0*. The value of *x* is unknown and therefore the value of the branch target is also unknown.

```
y := 0
if x > 0
    y := 7
```

Data dependencies can be grouped into the three categories: flow dependence (or true dependence), antidependence, and output dependence [85]. These are summarized below:

- Flow dependence is concerned with values flowing from one statement to another statement. Consider the statements S_1 and S_2 below:
 $S_1 : A = X + Y$
 $S_2 : B = A + Z$
 Here, S_2 is dependent on the assignment of A in S_1 - or put in another way, the value of A flows from S_1 to S_2 . Hence, S_1 must be executed before S_2 .

- Antidependence is concerned with statements where values used in one statement are overridden in proceeding statements. Consider the statements S_1 and S_2 below:

$S_1 : A = X + Y$

$S_2 : X = B + Z$

In the above example, the value of X is used in S_1 and overridden in the proceeding statement S_2 . Again, S_1 must be executed before S_2 .

- Output dependence is concerned with statements assigning to variables followed by statements assigning to the same variables which can result in variables containing incorrect values. Now, consider the statements S_1 , S_2 , and S_3 below:

$S_1 : A = X + Y$

$S_2 : C = A + Z$

$S_3 : A = B + Z$

Executing S_3 before S_1 may result in an incorrect value of A and therefore S_3 is dependent on S_1 .

To break dependencies among tasks, some sort of transformation needs to be performed. Consider the output example above. In this example, variable renaming [32] can be applied to break the dependency. The variable A in Statement S_1 and S_2 can be renamed to A' resulting in the following statements:

$S_1 : A' = X + Y$

$S_2 : C = A' + Z$

$S_3 : A = B + Z$

Clearly, S_3 is no longer dependent on S_1 . Thus, the dependency has been broken. This would probably not yield a significant improvement in performance since only one computational step has been removed. However, imagine the three statements embedded into a loop consisting of n iterations. In this case, the improvement is potentially n less computational steps.

Inferring that a variable carries a constant value, can also be used to break a dependency. Consider Listing 2.4. Inferring the value of x to be constant, would break that dependency since the value of the target of the branch is then known ahead. The constant value could for instance be 0 resulting in the code in Listing 2.4. Ergo, the branch will never be taken and the code can be reduced to the single computational step: $y := 0$.

Listing 2.4: Code example of breaking control dependency. The value of y depends on the branch target, a constant expression, $0 > 0$. The branch will therefore never be taken.

```
y := 0
if 0 > 0
    y := 7
```

2.4 Computer Architecture

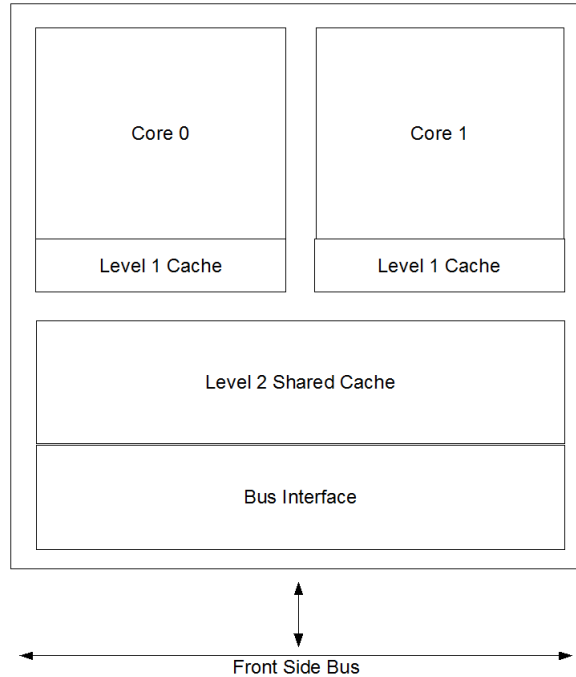
With concurrency identified, parallelism can be achieved by means of a parallel computer. In this section, computer architectures of parallel computers and superscalar processors are presented.

A traditional computer consists of one processor (processor and processing units are used interchangeably from now), whereas a parallel computer consists of multiple processors. There are two main parallel computer categories: multicomputers and multiprocessors [107].

For a multicomputer, multiple interconnected computers are operating together on a single problem [107]. The processors located on each computer interact by passing messages to each other [89].

A multiprocessor is an architecture where multiple processors are used within a single computer [107]. A special kind of multiprocessor are symmetrical multiprocessors (SMPs) (also known as centralized multiprocessors [89], shared memory multiprocessors, and uniform memory access (UMA) systems [32]). For such systems, access to a single global memory is shared among the processors [89]. As mentioned earlier, the current trend of several chip makers is to release multicore architectures (also known as chip multiprocessors (CMPs)) [7]. Multicore architectures consist of multiple processors being placed on a single chip [7]. In Figure 2.5, an example of such an architecture is shown.

Figure 2.5: CMP example - a simplified version of the Intel®Core™Duo processor architecture based on illustrations in [57]. In this architecture, each core has a level 1 cache and both cores share a level 2 cache. Further, the components are interconnected via a bus interface.



Another way to achieve parallelism, is by applying superscalar processors. Superscalar processors are able to execute two or more instructions per clock cycle [32]. One kind of such processors are processors adopting simultaneous multithreading (abbreviated SM [101] and SMT [34]). This approach consists of allowing multiple independent threads to send instructions to multiple functional units of a superscalar processor within a single clock cycle [101].

One must be aware that the architectures are not mutually exclusive but can be combined. A concrete example of such is the Intel®Core™i7 processor [53] where CMP and SMT is combined. Specifically, the processor consists of four cores with each core having two threads.

2.5 OS Support

The next step is communicating with the parallel computer which can be achieved through an OS. The two basic parallel processing methods supported by an OS are multiprocessing and multithreading.

Multiprocessing concerns creating processes where each process is assigned a process identifier (PID). Each of these processes may execute different programs. While processes usually are independent they may communicate with each other using interprocess communication or shared memory areas supported by the OS [32].

Multithreading concerns creating threads, where a thread differentiates from a process in that it is added to an existing process instead of starting a new process. A process is actually started with a single thread of execution and throughout its duration it can add or remove threads. Further, all threads in a process share memory space [32].

2.6 Parallel Programming

With OS support, parallel programming can be performed. To do parallel programming, one must make use of a parallel programming language. Such a parallel programming language is an instance of a programming model where this programming model must support a programmer in balancing productivity and implementation efficiency [13]. According to [13], opacity and visibility are the keys to achieve such a balance:

- Opacity makes the underlying computer architecture transparent for the programmer. Therefore, the programmer need not learn computer architecture dependent details and the programmer productivity increases.
- Visibility makes the details of the underlying computer architecture visible to the programmer. This means the programmer is able to analyze performance constraints of an application based on the design of the computer architecture.

The main issue here is achieving performance while raising the abstraction level.

Another important issue of a parallel programming model is the concurrency model. According to Joe Armstrong, the concurrency model of many programming languages is the same as the concurrency model of the underlying OS. This means that a concurrent program written in such a programming language executing on one OS may have different semantics compared to running the same program on a different OS [11]. Based on this, he argues that “...concurrency should be a property of the programming language and not a property of the...” OS. Further, he believes that “...the only difference in behavior in

moving a concurrent program from one machine to another is that the program will run faster on a faster machine etc; otherwise there should be no differences in behavior which depend upon the operating system.”.

2.7 Parallelizing Compilers

Obtaining parallelization manually is a demanding task. Instead, this procedure can be automatic¹ by using parallelizing compilers. Parallelizing compilers are compilers which transform an existing program to run efficiently on a parallel architecture [16].

For a compiler to automatically identify parallelism, the compiler should be able to do some form of dependence testing [85]. This way, the compiler can automatically extract threads², possibly preceded by transformations to break dependencies.

2.8 Performance

Executing a solution to the same problem using a parallel computer instead of a traditional computer should result in a performance improvement³. Otherwise, the extra computing capabilities are wasted. This improvement is measured by the speedup factor [107] as shown in Equation 2.1, where t_s = single processor execution time with the best sequential algorithm and t_p = parallel computer execution time with p processors and n data items:

$$S(p, n) = \frac{t_s}{t_p} \quad (2.1)$$

Usually, the maximum speedup possible is linear speedup, which means $S(p, n) = p$. Linear speedup can be achieved when a computation is dividable into processes with the same duration where each process is assigned to a separate processor without requiring extra overhead [107]. In some instances, superlinear speedup may be achieved, which means $S(p, n) > p$. However, this is usually due to applying a suboptimal sequential solution, a computer architecture feature which prefers parallel formation, or extra memory in the multiprocessor system compared to the single processor system [107].

For a theoretical analysis, the speedup factor can be measured based on computational steps [107] as shown in Equation 2.2, where s_s = number of computational steps using a single processor and s_p = number of parallel computational steps:

$$S(p, n) = \frac{s_s}{s_p} \quad (2.2)$$

Another equation applicable for theoretical analysis has been derived from a paper of Amdahl where this paper consists of arguing the validity of the single processor [6]. This equation has been coined “Amdahl’s law” and is shown in Equation 2.3, where r_p = the parallel portion of a program, $1 - r_p$ = the sequential portion of a program, and p = number of processors.

¹At least to some extent.

²- or whichever constructs are used to realize parallelism.

³At least when assuming homogeneous processors for both computers.

$$Speedup = \frac{1}{1 - r_p + \frac{r_p}{p}} \quad (2.3)$$

It illustrates well the fact that the less the parallel portion of a program is the less the speedup will be when increasing the number of processors.

However, Amdahl's law consists of a function too steep according to Gustafson whom has skeptically reevaluated Amdahl's law [47]. Some results which Gustafson achieved indicated that this function is incorrect. For instance, achieving speedup factors between 1021 and 1016 on a 1024-processor computer system for applications where the sequential part was 0.4% to 0.8%. For comparison, applying Amdahl's law, the speedup factors should be between 201 and 101. Instead of a steep function, Gustafson gives the alternative speedup function with a more moderate slope shown in Equation 2.4, where p = number of processors and s = the sequential portion of a program.

$$Speedup = p + (1 - p) * s \quad (2.4)$$

Interestingly, when applying Gustafson's speedup function for the parameters presented above (sequential parts and number of processors), the speedups computed are 1020 where $s = 0.4\%$ and 1016 where $s = 0.8\%$ and therefore consistent with the real speedup factors.

Efficiency is another measure of performance. This concerns with how much processors are used during a computation [107] and is shown in Equation 2.5 based on [107] and [51].

$$E(p, n) = \frac{S(p, n)}{p} = \frac{t_s/p}{t_p} \quad (2.5)$$

An alternative definition of efficiency is given in [51] with respect to a theoretical parallel machine instead of a real sequential machine. This definition is shown in 2.6, where t'_p is the time required for a theoretical parallel p -processor machine.

$$E(p, n)' = \frac{t'_p}{t_p} \quad (2.6)$$

2.9 Scalability

Scalability can refer to both architecture (or hardware) scalability or algorithmic scalability [107].

A system which is architecturally scalable is a system with a hardware design where increasing the size of the system increases performance.

An algorithmically scalable system applies a parallel algorithm where increasing the number of data items results in a low and bounded increase in computational steps.

Chapter 3

Parallel Processing Programming Models

For programming parallel computers, there are two dominant programming paradigms: the shared memory model and the message passing model [19, 23].

3.1 Shared Memory

In the shared memory model, there is a global address space which each processor has direct access to [19]. This means, a processor can directly load or store any shared address [29]. Through these shared addresses, or variables, the parallel executing program segments can communicate [19].

The vendors of shared-memory systems have created their own proprietary extensions to languages such as C and Fortran for the development of parallel software causing an absence of portability. This absence of portability have resulted in many developers adopting a portable message passing model (presented in the next section) instead [29].

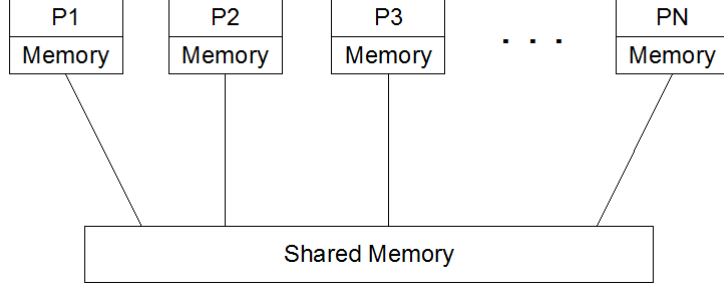
As a computation model for the shared memory model, the parallel random access machine¹ (PRAM) is available. The PRAM is the simplest generic model of parallel computers [5] and widely used [15, 27, 28]. A PRAM consists of p processors, each containing a private local memory, and a shared global random access memory (illustrated in Figure 3.1). It is assumed the processors work synchronously and any processor can access any memory cell in unit time [3, 14, 28, 30, 39, 43, 95, 40, 62].

There are several variants of PRAMs which are usually distinguished between using the rules for reading and writing as shown below:

- Exclusive read (ER): concurrent reading of a shared memory cell is forbidden.
- Concurrent read (CR): concurrent reading of a shared memory cell is allowed.

¹For a more detailed introduction to PRAM than provided here, the reader is referred to [40].

Figure 3.1: Illustration of PRAM (inspired by illustrations in [15] and [95]).



- Exclusive write (EW): concurrent writing to a shared memory cell is forbidden.
- Concurrent write (CW): concurrent writing to a shared memory cell is allowed.

[62] has shown that these variants do not vary much in computation speed. Still, a CREW is strictly more powerful than a EREW PRAM, whereas a EREW PRAM is strictly less powerful than a CRCW PRAM.

For a CRCW PRAM read- and write-conflicts can occur [62] and a rule for resolving concurrent writing needs to be applied [99]. Several possibilities exist to resolve such conflicts. Some of these are summarized below:

- The minimum model: each processor is assigned a priority; if multiple processors attempts to write to the same shared memory cell, then the processor with the highest priority succeeds [39].
- The arbitrary model: if multiple processors write to the same shared memory cell, an arbitrary processor will succeed [39].
- The common model: if multiple processors are writing a common value, then these processors are allowed to write to the same shared memory cell [39].

An alternative solution is to apply more restrictive variant of PRAM, i.e. a EREW PRAM or a CREW PRAM [62].

PRAMs are considered as the most powerful parallel computation models in the theory of parallel computation. Because the interprocessor communication need not be specified, such models are relatively comfortable to program. However, PRAMs are not realistic from a technological perspective since large shared memory machines can only be constructed at the cost of an access to shared memory which is very slow [30].

3.2 Message Passing

Before diving into the details of message passing, the π -calculus is shortly summarized to draw parallels.

The π -calculus is a process calculus [103] consisting of two fundamental concepts, which are processes (sometimes called agents [88]) and channels (also called names or ports [88]). Processes compute in parallel and exchange data by communicating through channels [87]. In fact, processes are constructed from channels and in its purest form all data is realized by channels [21].

Derived from an analysis of [17], [1], [49], [20], [92], and [25], usually a set of names for the channels and a set of variables are assumed from which processes are composed according to a given syntax. An example of a simplified syntax is given below based on [87] where the set of channel names range over c, d, \dots and the set of variables range over x, y, \dots :

- $P ::= \bar{c}[x_1 \dots x_n].P$; Send $x_1 \dots x_n$ along c , then become P . This communication is synchronous, i.e. execution is prevented for P until the communication on c has finished [103].
- $P ::= d(y_1 \dots y_n).P$; Receive $y_1 \dots y_n$ along d , then become P .
- $P ::= P|Q$; Execute P in parallel with Q .
- $P ::= (vc : T)P$; Create a new channel which carries type T , call it c in P .
- $P ::= 0$; Do nothing.

The syntax shown above relates to the polyadic π -calculus which is a generalization of the monadic π -calculus. The basic difference between these two is that in the polyadic form input and output is polyadic, i.e. in the form of tuples, whereas in the monadic form there is just a single input or single output usually denoted by $\bar{a}b$ where b is sent along a and $a(x)$ where x is received along a [92].

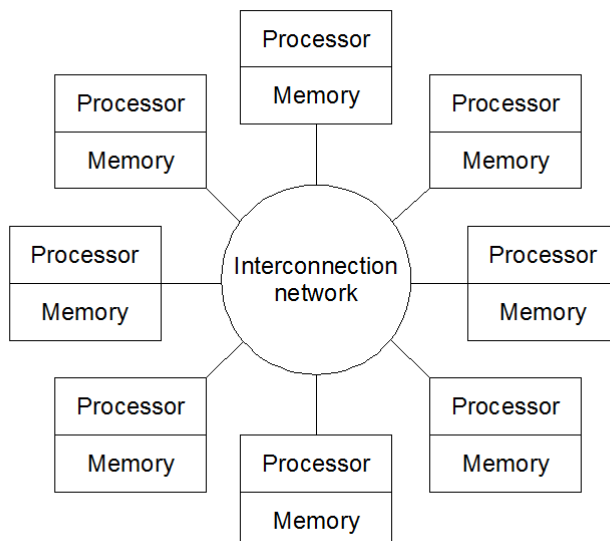
An example of a process composition is $\bar{c}[x_1, x_2].P_1|d(y_1, y_2).P_2$ which corresponds to a process consisting of two processes which run in parallel. The left hand process sends x_1, x_2 along c and then becomes P_1 while the right hand process receives y_1, y_2 along d and then becomes P_2 .

The literature of π -calculus contains many variations of syntax used for input and output [103]. For instance, four different syntax are used in respectively [87], [103], [83], and [22]. Furthermore, many versions of the π -calculus exist [1], including typed versions, e.g. in [20], asynchronous versions, e.g. in [83] and [49], and as mentioned above monadic and polyadic versions.

In the message passing model, data is divided and distributed to other processes as messages. The processes receiving a message, unpack the message, perform some work, and then sends back the result or pass along the results to other processes [32]. This form of communication is very similar to the communication between processes within the π -calculus.

The underlying hardware is assumed to be a collection of processors where each processor has its own local memory to which it has direct access. Through an interconnection network, message passing between processors is supported. Thereby, one processor gain indirect access to another processor's local memory [89]. This is illustrated in Figure 3.2. Due to the interconnection network, an implicit channel exists between each pair of processors [89].

Figure 3.2: Illustration of assumed underlying hardware of message passing (from [89]).



Often, the message passing model is referred to as the assembly language of parallel computers [66]. As a programmer, one gets the ultimate responsibility. If the resulting performance of an implementation is unsatisfactory, then oneself is to blame. The compiler is not aware of the parallel aspects of the program [32]. The π -calculus is rather low-level and can therefore also be seen as a kind of assembly language.

The basic routines of message-passing are *send* and *receive*, where *send* is placed in the source process creating the message and *receive* is placed in the destination process collecting the messages which are sent [107]. Again, the π -calculus is similar. The *send* could be viewed as an *output* process and the *receive* could be viewed as an input process.

3.3 Relation to Computer Architecture

The reader should be aware that the assumed underlying hardware of a parallel processing programming model does not need to be that specific hardware in reality. A mapping between the assumed underlying hardware and the hardware in reality can be performed. For instance, the message passing model can be mapped into SMP. This is for instance the case for SMP Erlang which is presented in Chapter 6.

Chapter 4

Compiler Theory

This chapter covers the relevant elements for the construction and discussion of GML. The purpose is not to provide an elaborate introduction to compiler theory. For an elaborate introduction, much existing literature is available, for instance [68] and [105].

The structure of this chapter follows the phases of the translation process of a compiler illustrated in Figure 4.1. The chapter ends with a summary of different compiler construction tools.

4.1 Scanner

The scanner performs lexical analysis by reading the source code represented as a stream of characters. Lexical analysis involves collecting sequences of characters for the recognition of tokens. Tokens are similar to words of a natural language. Along with the recognition of tokens, other tasks may be performed by a scanner, such as entering identifiers into a symbol table and entering literals into a literal table. A symbol table stores information related to identifiers, such as functions and variables. A literal table stores constants and strings which are used in a program [68].

Based on a simple example in C shown in Listing 4.1, the job of a scanner is exemplified in Listing 4.2.

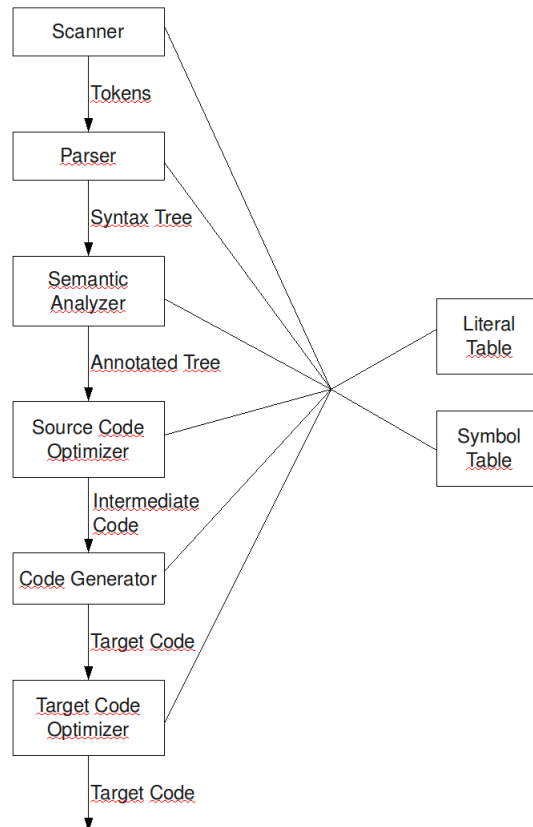
Listing 4.1: Example of source code in C. Inspired by [63].

```
a = 10 * 15.5 f
```

Listing 4.2: Example of scanning. Notice, spaces are ignored in Listing 4.1. The recognized tokens should be understood as follows: character sequence : token type.

a	:	identifier
=	:	assignment
10	:	number
*	:	multiply operator
15.5 f	:	number

Figure 4.1: Illustration of the phases of the translation process of a compiler (taken from [68] with small adjustments), i.e. scanning, parsing, semantic analysing, source code optimization, code generation, and target code optimization. In addition, it is shown that these phases use literal tables and symbol tables.



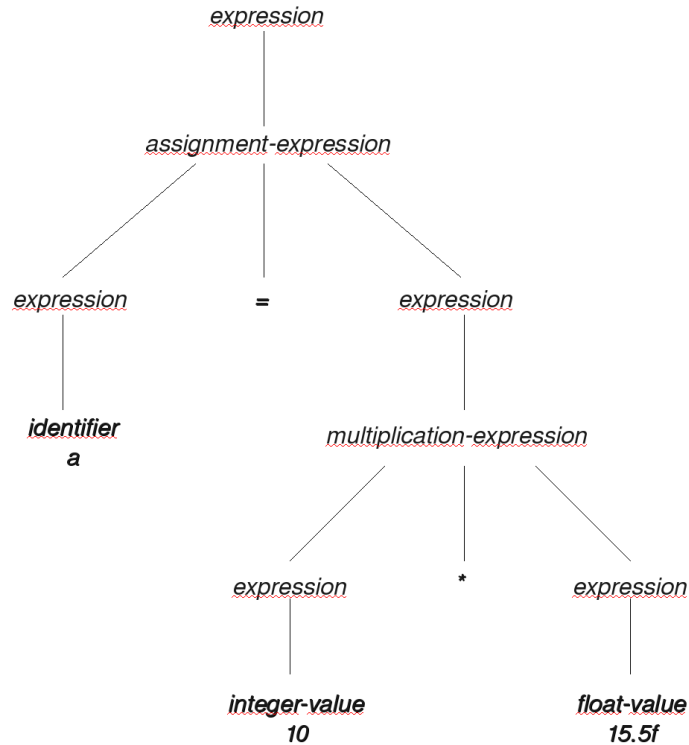
4.2 Parser

The tokens a scanner produce are passed on to a parser. From these tokens, the parser performs a syntax analysis. The syntax analysis consists of determining the structure of a program. More specifically, the structural elements and their relationships are determined. The results of this phase are commonly represented using a parse tree of a syntax tree [68].

To exemplify this, consider the source code presented in Listing 4.1. From a top-down perception, the source code consists of a single structural element, called an expression. This expression is a specialized expression called an assignment expression. The assignment expression consists of a left hand side expression and a right hand side expression with “=” as the separator. The left hand side expression “a” is a specialized expression called an identifier. The right hand side expression is a specialized expression called a multiplication

expression. The multiplication expression consists of a left hand side expression and a right hand side expression with “*” as the separator. The left hand side expression “10” is an integer value token. The right hand side expression “15.5f” is a float value token. The above structural composition is represented as a parse tree in Figure 4.2.

Figure 4.2: An example of a parse tree.



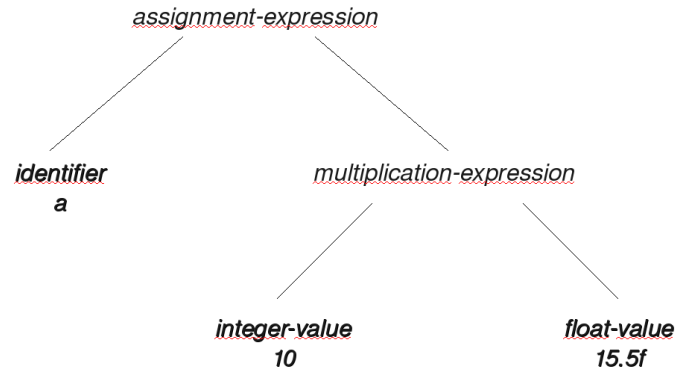
Though parse trees are useful for syntax visualization, they are inefficient due to the representation approach. Instead, syntax trees (also known as abstract syntax trees¹) can be used. Syntax trees are basically compressed parse trees [68]. For an example of a syntax tree, see Figure 4.3. Juxtaposing Figure 4.2 and Figure 4.3 illustrates the compression: many of the nodes are no longer part of the structure, including some token nodes. The nodes removed are considered unnecessary. For instance, the token node “*” in 4.2 is redundant since it is known that the expression is a multiplication expression.

4.3 Semantic Analyzer

The syntax tree (or parse tree) is passed on to the semantic analyzer. The semantic analyzer analyzes the semantics (or meaning) of a program producing additional pieces of information called attributes. These attributes are often used to annotate the tree or are entered into a symbol table [68].

¹The meaning of abstract here is that syntax trees introduce further abstraction than parse trees [68].

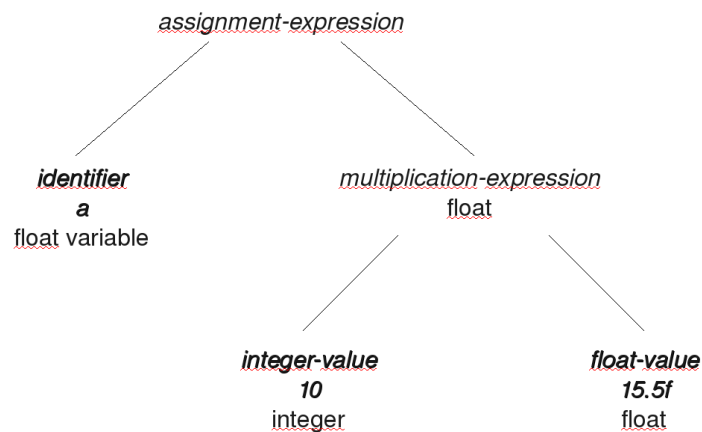
Figure 4.3: An example of a syntax tree.



Some of the semantics are part of a program's runtime behavior. Other semantics are determined before program execution. The latter is called static semantics. Static semantics are used when the syntax of a program is insufficient to express language features. Typical examples of static semantics are declarations and type checking [68].

Consider the source code in 4.2 again. Before the semantic analysis of that expression is performance, other analysis is performed, such that deriving that "a" is a variable of type float. Using these attributes, the tree is annotated resulting in the syntax tree presented in Figure 4.4.

Figure 4.4: An example of an annotated syntax tree.

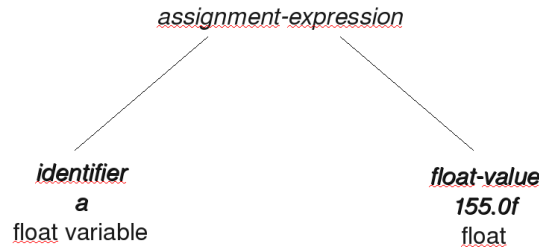


4.4 Source Code Optimizer

After the semantic analysis, source-level optimization techniques² can be applied. The result of a source code optimizer is intermediate code (also known as intermediate representation or simply IR). The kind of optimizations and placement in the compilation process of optimizations vary among compilers [68].

In relation to Figure 4.4, the following optimization can be performed. The multiplication expression can be statically reduced from “10 * 15.5f” to the constant float ‘155.0f’ (referred to as constant folding [68]). This can be performed directly on the tree. Specifically, the right hand side expression of the assignment expression is collapsed to the constant float, as illustrated in Figure 4.5.

Figure 4.5: An example of an optimized annotated syntax tree.



4.5 Code Generator and Target Code Optimizer

Using the intermediate code (IR), a code generator generates the target code. From this phase, the specific target machine becomes a more significant factor. For instance, the instruction set sets limitations and decisions about representation of integers and floating-point data types must be done [68].

Following code generation is target code optimization. During this phase, the target code generated by the code generator is potentially improved. For instance, different addressing modes can be chosen for optimization and slower instructions can be replaced by faster instructions [68].

4.6 Compiler Construction Tools

To assist the construction of compiler, a great number of tools are available. A summary of a few interesting tools is presented in this section.

Lex [67] in conjunction with Yacc [59] can be used to generate a compiler. Lex can generate programs for the lexical analysis whereas Yacc can generate programs for the syntax analysis. The generated source files are C programs.

²According to [68], this name is incorrect since these techniques rarely result in truly optimal target code but instead only efficiency improvements. They propose the name code improvement techniques as a replacement.

Similar to Lex and Yacc, Flex [86] and Bison [31] can be used to, respectively, generate C programs for the lexical analysis and C programs for the syntax analysis.

There are also tools which generate compiler programs in languages other than C. Among these is JavaCC [58]. As the name implies, the generated programs are Java [44] programs.

Another tool for generating Java Programs is ANTLR [8]. A GUI-based development environment is available for constructing ANTLR grammars called ANTLRWorks [9]. According to [9], ANTLRWorks speeds up development.

Chapter 5

Related Work

There is much work related to this project. In this chapter, some of that related work is summarized.

In [41], a comparison of implicit- and explicit parallel programming is made. For implicitly parallel programming, SISAL was used, and for explicitly parallel programming SR was used. The performance of these language was compared to implementations in C. They make the three conclusions presented below.

Due to the fact that the optimizing SISAL compiler (*osc*) is mature and well-developed, they do not expect improvements of *osc*. Though *osc* may not represent the limit of implicit parallelism, it indicates that this area is not approaching new improvements using similar approaches.

The biggest drawback of SR is performance. However, they expect improvements for SR. This is because some of the features implemented in *osc* has not yet been implemented in SR.

SISAL is good for only limited problem domains, mostly loop-parallel applications. For other applications, such as interactive applications, SISAL is less useful.

Intel® is currently researching on their data parallel environment called Ct [26]. Ct is a data parallel environment built on C++. Hence, it has predictable syntax. However, the semantics and performance provided are different. The data parallel capabilities of Ct, are realized by means of standard C++ templates. Higher level abstractions are used in order to support developers in building applications which scale across CMPs with hundreds of cores. Ct will become beta late 2009.

Experimental results [42] comparing Ct to sequential C have shown impressing results. Even superlinear speedups has been achieved. However, the experiment consisted mostly of relatively isolated tests, such as vector addition and square root calculation. Therefore, the representativeness of the results in larger program contexts might be limited.

The C++ parallel graph library Parallel BGL [46] applies generic programming for graph computations. By using generic programming, both flexibility and efficiency are achieved. Parallel BGL is adaptable to different communication models and data structures without compromising efficiency. Algorithms

and data structures for both distributed- and parallel graph computations are provided by Parallel BGL.

Experimental results of Parallel BGL performing on different graph models show in general good efficiencies and scalabilities. In the future, the authors are interested in looking into abstractions for supporting high-performance parallelism on shared-memory machines in relation to parallel BGL [46].

In [69], inter-relationships between graph problems, software, and parallel hardware are presented in the current state of the art. They argue that some properties of graph problems makes it difficult to solve these problems with the current computational problem-solving approaches. These properties of graph problems are: data-driven computations, unstructured problems, poor locality, and high data access to computation ratio.

They present hardware and software related challenges to address in order to map parallel graph algorithms to hardware. These challenges are: task granularity, memory contention, load balancing, simultaneous queries, and software development [69].

They conclude, among others, that massive multithreading seems an interesting approach for parallelizing graph computations due to a potential for excellent scalability and performance. In general, they are optimistic about the future despite the challenges regarding graphs and machines [69].

A portable implicitly parallel programming language for taking advantage of task-level concurrency called Jade is presented in [91]. A Jade program is initially written following an imperative serial paradigm. After this, Jade provides constructs for declaring how data access is performed in different parts of the program. With this data access information, concurrency is automatically extracted and the program is mapped towards the target machine. The semantics are kept intact during the parallel execution of the program.

There are implementations of Jade for the platforms: SMP, homogeneous message passing machines, and heterogeneous multicomputers [91].

They find that software development was simplified by keeping the serial semantics intact. They further find that a significant advantage is provided by using the data access information for parallelization instead of traditional control-based solutions. Finally, they achieved good performances for many applications on different hardware platforms with only little programming overhead. However, for some programs the results were less satisfactory. With improvements in the Jade implementation, some of these applications worked well. Other applications were better expressed in other languages [91].

Chapter 6

The Erlang Programming Language

The Erlang programming language is designed for concurrency oriented programming based on message passing.

The motivation behind the language model is based on the observation that the real world is concurrent [11]. Further on the intuitive perspective that, if a language designed for writing concurrent applications is used, then development will become much easier when developing such applications [12].

An interesting aspect of Erlang is the naming. One would intuitively think that it stands for “Ericsson Language”. Actually, Erlang is named for A. K. Erlang a Danish mathematician [104].

Erlang is typeless and pattern matching is used for binding variables and selecting functions [106]. Further, Erlang has single assignment variables. This means a variable can only be assigned once and reassignment of a variable results in an error [12].

In Erlang, the concurrency belongs to the programming language instead of the operating system. Parallel programming is achieved by modeling the world as sets of parallel processes which can only interact by exchanging messages and thereby no memory is shared [12]. To be more specific, there are primitives for spawning processes, sending messages to processes, and receiving messages [104].

An Erlang program may consist of thousands to millions of lightweight processes which can run both on a single processor or on a parallel computer [12]. A process is lightweight if it is created and destroyed using only very little computational effort [11].

A process is created, or spawned, using the *spawn* primitive. This is shown in Listing 6.1, where a new concurrent process is created which evaluates *Fun*. The *spawn* primitive returns a PID which can be used to send messages to the process [12].

Listing 6.1: Process creation syntax in Erlang

```
Pid = spawn(Fun)
```

Any data value can be used as a message and compression techniques are

used to minimize bandwidth requirements for transmitting values [104]. A data value may be either a constant, compound term, or a variable, where a constant is either an atom, float, PID, or integer [106]. Thus, sending arbitrary complex data values is both trivial and efficient [104].

The message passing in Erlang works as follows. Messages are not sent directly to processes. Instead, each process has an associated mailbox to which messages are sent [12]. Messages are in this mailbox queued in arriving order [111]. For this procedure, a primitive is available which is syntactically shown in Listing 6.2, where the primitive `!` is called the send operator and this operator returns the message itself [12].

Listing 6.2: Message send syntax in Erlang

```
Pid ! Message
```

In Listing 6.2, the message *Message* is sent to the process with identifier *PID*. Sending a message is non-blocking at the sender [12].

To receive a message, or rather examine the content in the mailbox, another primitive is available. This primitive is syntactically shown in Listing 6.3.

Listing 6.3: Message receive syntax in Erlang

```
receive
  Pattern1 [when Guard1] ->
    Expressions1 ;
  Pattern2 [when Guard1] ->
    Expressions2 ;
  ...
end
```

When a message is arrived at a process, the message is first matched against *Pattern1*, possibly with *Guard1*. If a match occurs, then *Expressions1* are evaluated, otherwise the message is matched against *Pattern2* and so forth. If no patterns are matched, then this message is saved in order to be processed later and the process will wait for the next message to be received [12].

With regards to the runtime environment of Erlang, there exists a version which takes advantage of SMP architectures. This version is called Symmetric Multiprocessing Erlang (SMP Erlang). SMP Erlang can be set to run with different numbers of schedulers. In some cases, having more schedulers than physical processors, increase throughput and makes a system behave better. These effects are however not fully understood and are currently undergoing research [12]. The strategy for SMP Erlang is to make SMP execution transparent for the programmer. There are some known bottlenecks. For instance, a single common run-queue is used which is predicted to become an issue as the number of processing units increase. This issue can be solved by applying separate run-queues per scheduler [70].

The scheduling can further be manipulated by changing priorities of processes. A process can be given one of the following three available priorities: normal, low, high, or max [82].

Before ending this section, let's compare the programming model of Erlang with the π -calculus (introduced in Section 3.2). The programming model is based on message passing which itself shares many properties with the π -calculus

as mentioned in Section 3.2. Hence, Erlang resembles the π -calculus in some ways, such as the communication approach of sending and receiving messages where one could perceive a specific pattern as a specific channel. However, the π -calculus is a mathematical definition of a calculus whereas Erlang is a concrete programming language.

Chapter 7

Graphs

Before starting the construction of the IGL, introductory material about graphs is needed.

First, the fundamentals of graphs is presented. Then, a number of general graph algorithms are presented followed by the graph application selected for this project.

7.1 Fundamentals

A graph $G = (V, E)$ where V is a set of vertices and E is a set of edges for which is true that $E \subseteq V \times V$. An edge between the vertices u and v is denoted by (u, v) [110].

A graph is either directed or undirected. In an undirected graph, for each $u, v \in V$ then $(u, v) \in E \Rightarrow (v, u) \in E$ [110].

Each edge has a source and a terminus. For (u, v) , the source is u and the terminus is v - except for edges in an undirected graph where both u and v are source and terminus [110].

The order of a graph is $N = |V|$ and the size of a graph is $M = |E|$ [48], where $|S|$ corresponds to the cardinality of the set S .

Two vertices $u, v \in V$ are adjacent if $(u, v) \in E$. The edge (u, v) is incident to its vertices u and v [48].

A graph may contain multiple paths. A path from u to v of a graph G corresponds to a sequence of n edges e_1, \dots, e_n of G where $e_1 = (x_0, x_1)$, $e_2 = (x_1, x_2)$, ..., $e_n = (x_{n-1}, x_n)$, $x_0 = u$, and $x_n = v$ [93]. The source and target in a path from u to v is respectively u and v .

7.2 Algorithms

Many graph algorithms have been developed. In this section, a few of these algorithms are presented.

Two commonly applied algorithms are breadth-first search and depth-first search algorithms. A breadth-first search algorithm fans out graphs by exploring each adjacent vertex before traversing further into the graphs [65]. On the contrary, a depth-first search traverses through a graph by recursively exploring the adjacent vertex u to a vertex v . This is performed until there are no adjacent

vertices to v and backtracking is performed to visit the remaining adjacent vertices v [93].

A common problem within graph theory is finding the shortest path in a graph. One known algorithm which aims to solve this problem is Dijkstra's algorithm. In short, this algorithm always chooses the closest vertex in a given graph. The time complexity of the sequential version is generally $O(|V|^2)$ [109]. A parallel version, for which the PRAM EREW model is assumed, runs $O(|V| \log |P|)$, where P is the number of processors [109].

7.3 Application

With a graph application, the APIs for IGL can be derived and GML can be designed. There are several applications of graphs. This section does not contain an exhaustive analysis of multiple applications since this would be a time wise expensive task. Instead, a single simple application recommended by the main supervisor of the author is presented.

The application is taken from [84] where Algorithm 1 is suggested as a simple solution to graph related issues ¹.

Algorithm 1 Graph application algorithm.

- 1: Calculate intial total entropy $H_{co_0}(G)$ and $H_{ce_0}(G)$
 - 2: **for all** nodes \in graph G **do**
 - 3: Remove node v_i , creating a modified graph G'
 - 4: Recalculate $H_{co_i}(G')$ and $H_{ce_i}(G')$, store these results
 - 5: Restore original graph G
 - 6: **end for**
 - 7: To solve the KPP-Pos problem, select those nodes that produce the largest change in graph entropy $H_{co_0} - H_{co_i} \geq \delta_1$
 - 8: To solve the KPP-Neg problem, select those nodes that produce the largest change in graph entropy $H_{ce_0} - H_{ce_i} \geq \delta_2$
-

¹For further details, the reader is referred to [84].

Chapter 8

Graph Library and Meta Language

In this chapter, Algorithm 1 is made more explicit for the extraction of the APIs for IGL. The IGL APIs and the more explicit algorithm are then used as the foundation for GML.

8.1 A More Explicit Algorithm 1

Some of the steps in Algorithm 1 contains implicit information. A more explicit algorithm is required to extract the APIs for IGL¹. The more explicit algorithm is presented in Algorithm 2 for which the definition of a graph G is assumed.

Algorithm 2 Graph application algorithm (Explicit). The algorithm is on purpose very explicit for an algorithm to better support the mapping between IGL and GML. The operation STORE X AS Y means store the value of the expression X in the storage location Y . I and J can carry any positive integer value.

- 1: Store table with fields $\langle v, H_{co}, H_{ce} \rangle$ as H_{all}
 - 2: Store δ_1 as I
 - 3: Store δ_2 as J
 - 4: **for all** nodes $v_i \in \text{graph } G$ **do**
 - 5: Store a copy of G as G'
 - 6: Remove node v_i from G'
 - 7: Add row $\langle v_i, H_{co_i}(G'), H_{ce_i}(G') \rangle$ to H_{all}
 - 8: **end for**
 - 9: Select v of the first δ_1 tuples of H_{all} sorted ascending by H_{co}
 - 10: Select v of the first δ_2 tuples of H_{all} sorted ascending by H_{ce}
-

¹For the interested reader, the extraction can be seen in Appendix A.

8.2 Graph APIs

In this section, the IGL APIs are inferred from Algorithm 2. The mapping² between Algorithm 2 and the IGL APIs are summarized in table 8.1.

Table 8.1: Mapping from Algorithm 2 to IGL APIs.

Lines	IGL API
Implicated in 1, 4, 5, 6, and 7	<i>graph_complete(N)</i>
Implicated in 1, 4, 5, 6, and 7	<i>graph_sparse(N)</i>
1	<i>table_new()</i>
4	<i>graph_nodes(G)</i>
5	<i>graph_copy(G)</i>
6	<i>graph_nodes_remove(G, V)</i>
7	<i>table_rows_add(T, R)</i>
7	<i>table_row_new(Cs)</i>
7	<i>graph_connectivity(G)</i>
7	<i>graph_centrality(G)</i>
9 and 10	<i>table_select(T, Fs)</i>
9 and 10	<i>table_first(T, X)</i>
9 and 10	<i>table_sort(T, S)</i>

The Erlang digraph module [37] is used for graph representation and sequentially supported operations. For the construction of complete graphs and sparse graphs, the approaches presented in Listing 8.1 and 8.2 are applied, respectively.

Listing 8.1: Construction of sparse graphs where N = the number of vertices in the graph G to construct.

```
G = new undirected graph
for each i in 1 to N
  add vertex i to G
  for each n in vertices of G
    add edge between i and n in G
```

Listing 8.2: Construction of complete graphs where N = the number of vertices and S = the number of edges each vertex should be connected to in the graph G to construct.

```
added_vertices = empty list
G = new undirected graph
for each i in 1 to N
  add vertex i to G
  verts to connect to =
    at most S vertices in added_vertices reversed
  for each n in verts to connect to
    add edge between i and n in G
  add i to added_vertices
```

²In Appendix B, a more elaborate mapping is presented.

8.3 Meta Language

For the construction of GML, ANTLRWorks is used. This is because this tool should speed up development and there is limited time available for this project. In the current hardware situation, many hardware platforms consist of a limited number of processors. Therefore, according to the discussion in 2.2, loop-iteration level parallelism is perceived as important for GML. In general, the objective is to automatically extract the same number of threads as the number of processors available.

8.3.1 Grammar

Clearly, GML should contain primitives corresponding to the APIs presented in Section 8.2. This mapping is presented in Table 8.2.

Table 8.2: Mapping from IGL APIs to GML primitives and their grammar.

IGL API	GML Primitive Grammar
<i>graph_complete(N)</i>	<code>#{N}</code>
<i>graph_sparse(N)</i>	<code>#{N ',' N/5}</code>
<i>table_new()</i>	<code>'[F (',' F)*]'</code>
<i>graph_nodes(G)</i>	<code>'NODES' 'OF' G</code>
<i>graph_copy(G)</i>	<code>'COPY' 'OF' G</code>
<i>graph_nodes_remove(G, V)</i>	<code>'REMOVE' V 'IN' 'NODES' 'OF' G</code>
<i>table_rows_add(T, R)</i>	<code>'ADD' R 'TO' T</code>
<i>table_row_new(Cs)</i>	<code>'< C (',' C)* >'</code>
<i>graph_connectivity(G)</i>	<code>'CONNECTIVITY' 'OF' G</code>
<i>graph_centrality(G)</i>	<code>'CENTRALITY' 'OF' G</code>
<i>table_select(T,Fs)</i>	<code>'SELECT' F (',' F)* 'OF' T</code>
<i>table_first(T,X)</i>	<code>'FIRST' X 'OF' T</code>
<i>table_sort(T,S)</i>	<code>'SORT' T 'BY' S</code>

In addition, a primitive to store variables (`'STORE' X 'AS' ID`) and enumeration of nodes in a graph (`'FOR' 'ALL' V 'IN' 'NODES' 'OF' G ... 'END'`) is required. A simplified grammar of GML is presented in Listing 8.3. This grammar both represents the lexical- and syntax analysis of the compilation process. Using this grammar, Algorithm 2 is realized by the GML source in Listing 8.4.

Listing 8.3: A simplified grammar of GML.

program	::= statements
statements	::= statement*
statement	::= storeStmt forAllStmt removeStmt addStmt selectStmt
storeStmt	::= 'STORE' value 'AS' Identifier
forAllStmt	::= 'FOR' 'ALL' Identifier 'IN' value statements 'END'
removeStmt	::= 'REMOVE' value 'IN' value
addStmt	::= 'ADD' value 'TO' value
selectStmt	::= 'SELECT' Identifier (',' Identifier)* 'OF' value
value	::= Identifier Integer graph nodes copy row entropy first sort
graph	::= '#' '{' value (',' value)? '}'
nodes	::= 'NODES' 'OF' value
copy	::= 'COPY' 'OF' value
row	::= '<' value (',' value)* '>'
entropy	::= ('CONNECTIVITY' 'CENTRALITY') 'OF' value
first	::= 'FIRST' value 'OF' value
sort	::= 'SORT' value 'BY' value
Identifier	::= ('a'..'z' 'A'..'Z') ('a'..'z' 'A'..'Z' '0'..'9' '\ ')*
Integer	::= '0' (('1'..'9') ('0'..'9')*)

Listing 8.4: General IPI of Algorithm 2 in GML. ? is either “#{N}” or “#{N,N/5}”. *I* and *J* are each replaced by a positive integer value in an application.

```

STORE ? AS G
STORE [Node, Conn, Cent] AS ents
STORE I AS thresh_conns
STORE J AS thresh_cents

FOR ALL v IN NODES OF G
  STORE COPY OF G AS G'
  REMOVE v IN NODES OF G'
  ADD <V, CONNECTIVITY OF G', CENTRALITY OF G'> TO ents
END

SELECT Node OF FIRST thresh_conns OF SORT ents BY Conn
SELECT Node OF FIRST thresh_cents OF SORT ents BY Cent

```

8.3.2 Parallelization

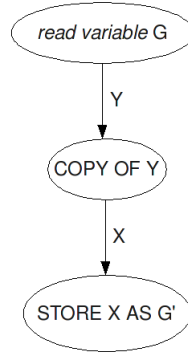
The parallelization of a GML application relates to the source code optimization phase of a compiler. The first step of parallelization of GML source involves

identification of concurrency. For this, a DDG is used to represent the dependencies among the different GML primitives part of a GML source. Partly, the grammar of GML is used to extract the dependencies. Partly, the semantics. For an example where the grammar is used, consider the GML source below:

`STORE COPY OF G AS G'`

In this case, the grammar itself for the STORE statement can be used to extract the DDG shown in Figure 8.1.

Figure 8.1: Illustration of grammar-based DDG-extraction. Notice, “*read variable G*” is implicit in the GML source.



For an example where both grammar and semantics are used, consider the GML source below:

`STORE [Int] AS ents`
`ADD <5>T0 ents`

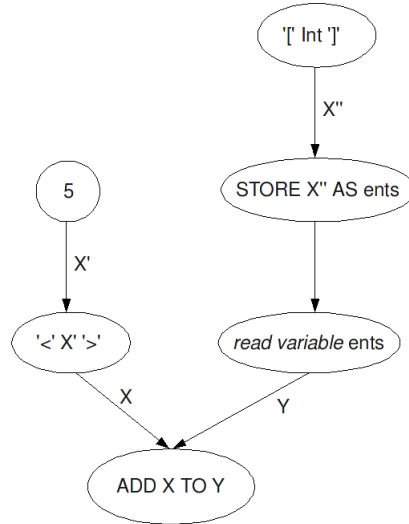
In this case, the grammar itself for the STORE and ADD statement is insufficient. The implicit “*read variable ents*” of the ADD statement is dependent on the STORE statement, since the STORE statement writes to the variable “ents”. This is illustrated in Figure 8.2.

During the extraction of the DDG, the entry points of an application are found. An entry point is a node which has no ingoing edges, that is, that program element has no dependencies.

The DDG and entry points are used to construct a data flow graph consisting potentially of four types of elements: threads, forks, joins, and parallel loops.

Threads correspond to serial code and are extracted following the pseudo code presented in Listing 8.5. Threads consist of a chain of statements where a purely sequential dependency exists. Specifically, the first program node of the chain has either zero (is an entry point) or multiple (marks a join) incoming edges. The last program node of the chain has either zero (is an exit point) or multiple (marks a fork) outgoing edges. If a node is not the first node of a chain, then it has one ingoing edge. If a node is not the last node of a chain, then it has one outgoing edge.

Figure 8.2: Illustration of both grammar- and semantics-based DDG-extraction.



Listing 8.5: Pseudo code for extracting threads.

```

function extract threads(nodes)
  threads = empty list
  for all n in nodes
    thread nodes = empty list
    extract thread nodes(n, thread nodes)
    thread = build thread from thread nodes
    append thread to threads
  return threads

function extract thread nodes(n, thread nodes)
  if n has one outgoing edge then
    add n to thread_nodes
    e = outgoing edge of n
    extract thread_nodes(terminus of e, thread nodes)

```

A fork is identified by a program node having multiple outgoing edges and one ingoing edge. A fork (potentially) marks the end of a thread and marks the start of multiple threads. The approach followed for the extraction of forks is presented by the pseudo code in Listing 8.6. Notice that the addition of fork extraction requires extension to the function *extract thread nodes*.

Listing 8.6: Pseudo code for extracting forks.

```
function extract thread nodes(n, thread nodes)
.
.
.
if n has multiple outgoing edges
and one (or zero) ingoing edges then
    extract fork(n)

function extract fork(n)
    thread starts = empty list
    for all outgoing edges e of n
        add terminus of n to thread starts
    threads = extract threads(thread starts)
    fork = build fork from threads and n
```

A join is identified by a program node which has multiple ingoing edges and one outgoing edge. A join marks the end of multiple threads and (potentially) the start of one thread. For extraction of joins, the approach illustrated using pseudo code in Listing 8.7, is followed. The addition of join extraction requires extension to the function *extract thread nodes* again.

A parallel loop is identified by program node which is annotated as a parallel loop. This approach is illustrated in Listing 8.8 using pseudo code. It is assumed that no loop carried dependencies exists among iterations of a GML loop.

Listing 8.7: Pseudo code for extracting joins. “inv” is short for “inverse”.

```

function extract thread nodes(n, thread nodes)
.
.
.
if n has multiple ingoing edges
and one (or zero) outgoing edges then
    extract join(n)

function extract join(n)
    thread ends = empty list
    for all ingoing edges e of n
        add source of n to thread ends
    threads to join = extract threads inverse(thread ends)
    fork = build fork from threads and n

function extract threads inverse(nodes)
    threads = empty list
    for all n in nodes
        thread nodes = empty list
        extract thread nodes inverse(n, thread nodes)
        thread = build thread from thread nodes inverse
        append thread to threads
    return threads

function extract thread nodes inv(n, thread nodes)
    if n has one ingoing edge and one outgoing edge then
        add n to thread_nodes
        e = ingoing edge of n
        extract thread nodes inv(source of e, thread nodes)

```

Listing 8.8: Pseudo code for extracting parallel loops.

```

function extract thread nodes(n, thread nodes)
.
.
.
if n is annotated as a parallel loop then
    parallel loop = build parallel loop from n
    return parallel loop

```

8.3.3 Target Code Generation

The language of the target code is Erlang. The target code is generated according to the constructed data flow graph during source code optimization.

Each thread corresponds to spawning a new process consisting of the code the thread constitutes. So, the multiple threads of a fork corresponds to spawning

multiple processes.

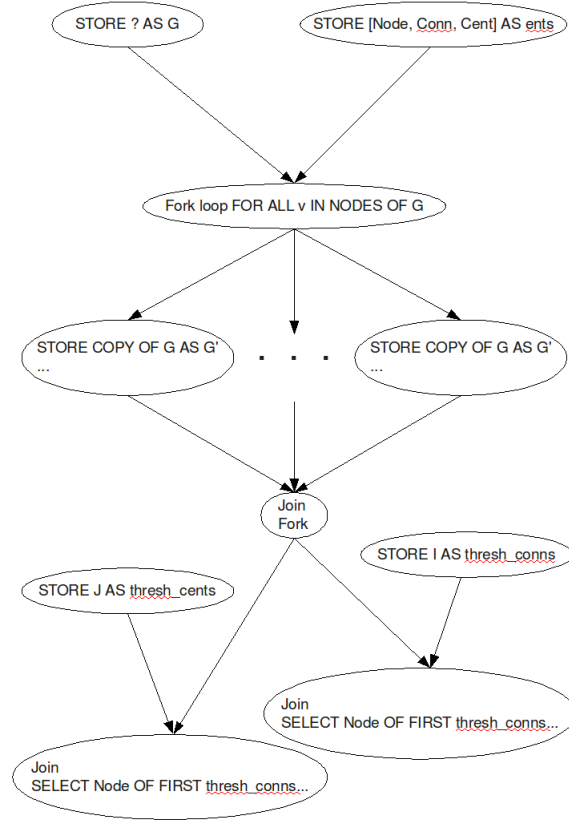
A join is translated into an process waiting for N messages from its N threads. When a join has received N messages, then if the join has a dependent thread that thread is spawned otherwise the join marks an exit point.

A parallel loop is essentially translated into a fork and a join. The fork spawns the threads which represent the iterations of the loop. Each thread is assigned to one or more of the iterations of the loop. The join then simply waits for the threads representing the iterations of the loop to send a message to it.

Since the parallelized GML program consists of one or more entry points and one or more exit points, an implicit fork for entry points and an implicit join for exit points are generated.

The compiler process described above results in a parallelized version of Algorithm 2. This parallelized version is illustrated in Figure 8.3 using GML primitives.

Figure 8.3: Illustration of a generalized and simplified approach of IPI (and EPI). I and J are each replaced by a positive integer value in an application.



8.3.4 Theoretical Improvements for Parallelization

Before realizing any implementations, let's consider the theoretical improvements of the parallelized version of Algorithm 2 - without theoretical improvements implementations do not make much sense. Assume the time complexities in Table 8.3 and that a STORE operation has the time complexity $O(1)$.

Table 8.3: Assumed time complexities for Algorithm 2 where n = number of nodes and r = number of rows. The assumption for *graph centrality*(G) is made based on the fact that ETS is used for graph representation in the Erlang digraph module. Notice, for other graph representations this assumption might be contradictive. For instance, if an adjacency matrix [45] was used then n^2 is more correct.

IGL API	O
<i>graph_complete</i> (N)	n^2
<i>graph_sparse</i> (N)	n^2
<i>graph_nodes</i> (G)	1
<i>graph_copy</i> (G)	n
<i>graph_nodes_remove</i> (G, V)	1
<i>table_rows_add</i> (T, R)	1
<i>table_row_new</i> (Cs)	1
<i>graph_connectivity</i> (G)	n
<i>graph centrality</i> (G)	n^3
<i>table_select</i> (T, Fs)	r
<i>table_first</i> (T, X)	r
<i>table_sort</i> (T, S)	r

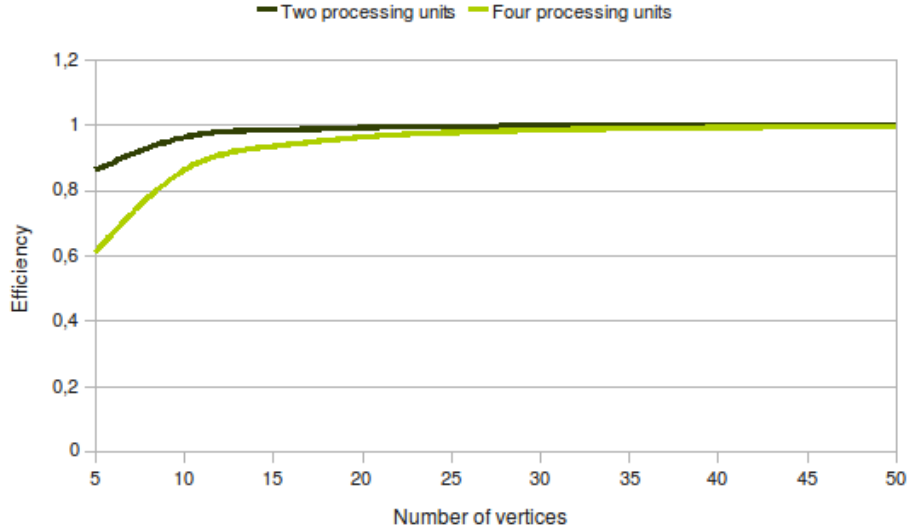
The time complexity for each iteration of the for all loop is $O(n+1+n+n^3+1)$ or in short $O(n^3)$. For the whole loop, it is $O(n^4)$. For line 1, it is n^2 . For lines 2 through 4 it is for each $O(1)$. For lines 9 and 10 it is for each $O(r^3)$. Thus, the time complexity of a serial version of Algorithm 2 is $O(n^4)$.

For the parallel version of Algorithm 2, there is some coordination involved. Assume that for the coordination of forking threads and joining threads the time complexity is kt where k = some constant and t = the number of threads. In addition, since it is the nodes of a graph which are distributed and reduced in relation to forks and joins, respectively, coordination is extended to $kt * n$. Then, the time complexity of the parallel version is $O(\frac{n^4}{p} + kt * n)$ where p = number of processing units. Bare in mind, that t is not necessarily equal to p . It is the maximum t which is considered for the time complexity. The maximum $t = p$ in the case of Figure 8.3.

Due to the above, theoretically, the parallel version has significant potential for improvements in practice compared to the serial version. However, linear speedup is not theoretically achievable due to mainly four factors. First, the task parallelism existing in the beginning of Figure 8.3 is limited to two parallel threads. Second, $kt * n$ reduces the speedup potential. Third, the task parallelism following the parallel loop is limited to two parallel threads. Thus, the optimal number of processing units seems to be 2. However, t is rather insignificant compared to $\frac{n^4}{p}$. Based on this, theoretically, there should be different efficiencies between using two processing units compared to using four processing units, but this difference is most likely insignificant.

When considering the additional factors, the coordination must be extended further. For simplicity, the coordination is extended to $kt * n^2$. Hence, the final time complexity is $O(\frac{n^4}{p} + kt * n^2)$. Figure 8.4 shows the theoretical efficiencies for using two processing units (TTE) and four processing units (FTE) where $k = 1$. The figure shows a proportional correlation between the number of vertices and the efficiency. An advantage of using two processing units instead of four processing units is also shown. It further shows that at some point linear speedup is almost achieved. However, linear speedup is most likely not realistic based on the discussion above. The results of executing the benchmark presented in the next chapter shall show whether linear speedup is achieved at some point or not.

Figure 8.4: Illustration of theoretical efficiencies according to the time complexity where $k = 1$.



8.3.5 Correctness

For verifying the correctness of the parallel implementations for the benchmark (presented in the next chapter), the outcomes for the parallel implementations are compared to the outcomes for the sequential implementations (presented in the next chapter). An outcome consists of a sequence of δ_1 (or δ_2) triples, with each triple having the form (v, c_o, c_e) , where v = the vertex, c_o = the connectivity entropy, and c_e = the centrality entropy. The sequence of the triples and the c_o s and c_e s for each triple of the outcomes are compared between the sequential implementation and each parallel implementation. If no differences between the sequence or the c_o s and c_e s of the sequential- and a parallel implementations exists, then the parallel implementation is perceived as correct.

All outcomes are presented in Appendix C. These outcomes show that no differences exist between the sequential implementation and each parallel implementation. Ergo, the parallel implementations are perceived as correct.

8.3.6 Limitations

GML is designed to satisfy the needs for parallelizing a GML implementation of Algorithm 2. Thus, the parallelization approach is most likely not applicable for different graph applications than the one considered in this project. For instance, some assumptions are made, such as assuming no loop-carried dependencies exist in a GML loop. Further, GML only has primitives for an implementation of Algorithm 2.

Chapter 9

Benchmark

The benchmark is centered around implementations of Algorithm 2. As mentioned before, for this algorithm the definition of a graph G is assumed. For the implementations, G is realized by either a complete graph G_c or a sparse graph G_s .

The graph orders (5, 10, 15, 20, 25, 30, 35, 40, 45, 50) are chosen to test algorithmic scalability for both G_c and G_s . These orders are denoted as O , o_i denotes the i^{th} order of O , and o denotes an order in O .¹ For each graph order, the speedup and efficiency is derived. The algorithmic scalability is evaluated as follows. Let s_i denote speedup for o_i . If speedup difference $s_{i+1} - s_i \geq 0$ then the implementation is considered algorithmically scalable when increasing the order from o_i to o_{i+1} . Otherwise, not scalable².

For this benchmark, each vertex v of G_s is connected to $o/5$ vertices distinct from v . So, in total the benchmark consists of 20 distinct test cases.

The following implementations are constructed: a single sequential implementation (SI) using IGL, a single explicitly parallelized implementation (EPI) using IGL, and multiple implicitly parallelized implementations (IPIs) using GML. There are multiple IPIs since these are compiled from different GML sources each containing different definitions of G (illustrated in Listing 9.1). The general IPI is the same as shown in Listing 8.4.

Listing 9.1: Different definitions of G in GML

Ex. 1: STORE #{10} AS G
Ex. 2: STORE #{25} AS G
Ex. 3: STORE #{40,8} AS G

To test architectural scalability and improve representativeness, the benchmark is executed on multiple different platforms. These platforms are listed below:

Duo Core Platform (DCP):

- Intel®Core™Duo Mobile Processor T9300 (duo core processor).

¹These orders were chosen due to interesting trends based on preliminary tests - especially for the lower numbers.

²Perceive 0 here as approximately 0, not exactly 0.

- Ubuntu 9.04, 64-bit.
- Erlang OTP R12B-5.
- 3.0 GB RAM.
- 2.5 GHz pr. processor.

Quad Core Platform (QCP):

- Intel®Core™Quad Processor Q6600 (quad core processor).
- Windows Vista™Ultimate, SP 1, 32-bit.
- Erlang OTP R13B.
- 3.0 GB RAM.
- 2.4 GHz pr. processor.

SMT Platform (SMTP):

- Intel®Pentium 4 SMT Processor (single SMT processor, two threads).
- Windows XP professional.
- Erlang OTP R12B.
- 2.0 GB RAM.
- 3.4 GHz pr. processor.

Intuitively, the optimal number of parallel threads are two, four, and two for respectively DCP, QCP, and SMTP based on the number of processing units available. Due to this, the IPIs are during compilation optimized according to these numbers of threads.

Each test case is timed using the *timer:tc* Erlang API [38]. The time resolution of *timer:tc* is microsecond, i.e. 10^{-6} seconds³ [73]. For improving representativeness and to reduce risks for bias, each test case is executed 20 times and reduced to a single number. According to [96], the arithmetic mean (AM) (shown in Equation 9.1) “can be used as an accurate measure of performance expressed as time.”. Due to this, AM is applied for the reduction of the measured execution times.

$$\overline{x}_a = \frac{1}{N} \sum_{i=1}^N x_i, N = \text{cardinality}. \quad (9.1)$$

For general perspectives, the derived speedups and efficiencies are reduced. Speedups and efficiencies are reduced by applying the harmonic mean (shown in Equation 9.2) because this is likely a more appropriate approach in order to limit the significance of potential exceptions.

$$\overline{x}_h = \frac{N}{\sum_{i=1}^N \frac{1}{x_i}}, N = \text{cardinality and } x_i > 0. \quad (9.2)$$

³- or simply a millionth of a second.

The criteria for success is simple⁴: at least, maintain performance improvements achieved so far. Lets define an approach to evaluate this.

According to [18], historically, the number of transistors on chips have double for each 18. month whereas the performance has doubled for each third year. They make the assumption that if the additional transistors are used for additional cores on chips then the doubling of cores should result in a speedup of about 1.4 for maintenance of the performance improvements so far. This corresponds to an efficiency of 0.75 and 0.49 for two and four processing units, respectively. These efficiencies shall mark the thresholds for maintenance of historical performance improvements for the respective number of processing units for this benchmark. These thresholds are used to evaluate the architectural scalability of the implementations, i.e. if the thresholds are exceeded for both two and four processing units then the implementations are considered architecturally scalable.

In the next chapter, IPI is compared to SI, EPI, respective thresholds for maintenance of historical performance improvements, and the theoretical efficiency presented in Section 8.3.4.

⁴That is, simple to perceive - not necessarily simple to achieve.

Chapter 10

Evaluation

The results, analysis, and evaluation of each benchmark execution is presented following the sequence of platforms: DCP, QCP, and SMTP. Each section is divided into G_c - and G_s -based benchmarks. The G_c - and G_s -based benchmarks are compared for each platform. DCP and SMTP are compared to TTE and QCP is compared to FTE. The chapter ends with an overall comparison.

During the chapter the proceeding short hand notation is applied:

- Arithmetic mean execution time = $\bar{\mu}$.
- Arithmetic mean execution time in seconds = T .
- Number of vertices = N .
- Speedup = S .
- Harmonic mean speedup = \bar{S} .
- Efficiency = E .
- Harmonic mean efficiency = \bar{E} .
- The threshold for maintenance of historical performance improvements = δ .
- Harmonic mean difference between theoretical efficiency and achieved efficiency = $\overline{\delta_E}$, where $x_i = abs(o_{i_t} - o_{i_a})$, o_{i_t} = theoretical efficiency for o_i , and o_{i_a} = achieved efficiency for o_i . This is an expression of how precise the theoretical efficiency is; the closer to zero, the more precise.

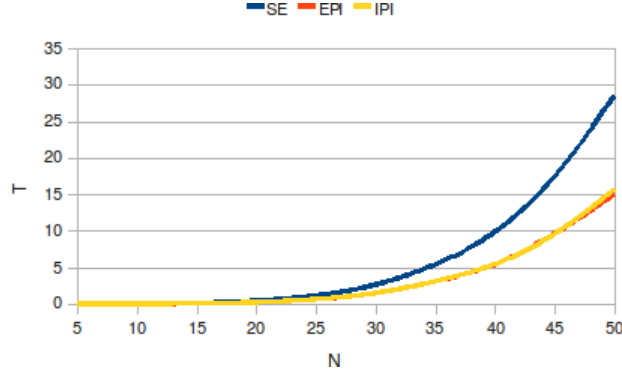
10.1 DCP

Below, the results, analysis, and evaluation of executing the benchmark on DCP is presented. The section ends with a comparison between the evaluation of the G_c - and G_s -implementations.

10.1.1 G_c

In Figure 10.1, the $\bar{\mu}s$ derived for SI, IPI, and EPI are illustrated. The results show in general a performance advantage for EPI and IPI against SI. The $\bar{\mu}s$ for EPI and IPI are similar¹.

Figure 10.1: Illustration of $\bar{\mu}s$ derived for SI, EPI, and IPI on DCP in relation to G_c .



The derived Ss for EPI and IPI are shown in Figure 10.2. These Ss are similar. A significant proportional correlation between N and S can be seen from $N = 5$ to $N = 20$. Hereafter, this relation, more or less, stagnates or at least becomes marginal. \bar{S} is 1.60 and 1.58 for respectively EPI and IPI showing a marginal advantage for EPI. EPI and IPI are both algorithmically scalable, especially from $N = 5$ to $N = 20$.

Figure 10.2: Illustration of Ss derived for EPI and IPI on DCP in relation to G_c .

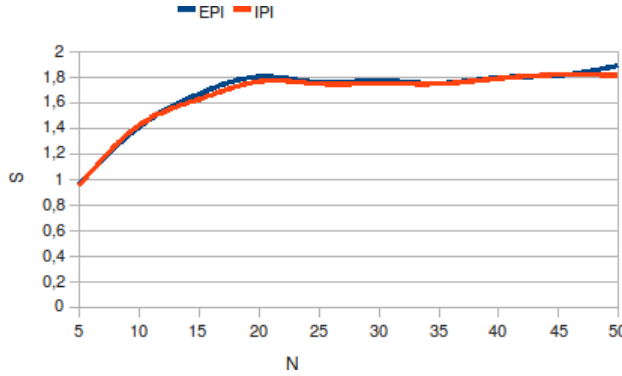
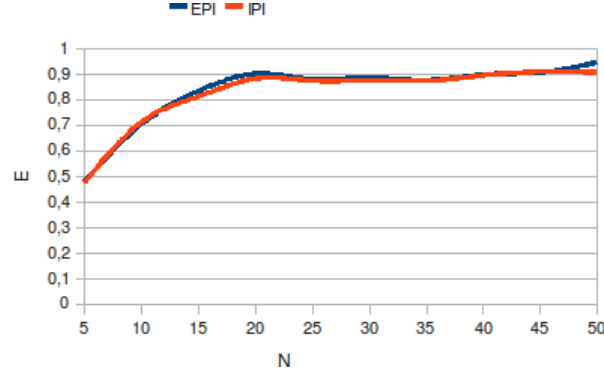


Figure 10.3 shows the derived Es for EPI and IPI. \bar{E} for EPI is 0.80 and for IPI 0.79. This exceeds δ . Therefore, for IPI, maintenance of performance is achieved for DCP in relation to G_c . Comparing the Es for IPI to TTE, then similar trends are shown. A proportional correlation till $N = 20$ is present. Hereafter, this correlation is much less significant or stagnates. Linear speedup

¹Illustrated by the clear overlap between EPI and IPI in Figure 10.1 making EPI difficult to see in many contexts.

is not achieved at any point for IPI. $\overline{\delta_E}$ is 0.13 which means there is a moderate difference between theoretical efficiency and achieved efficiency.

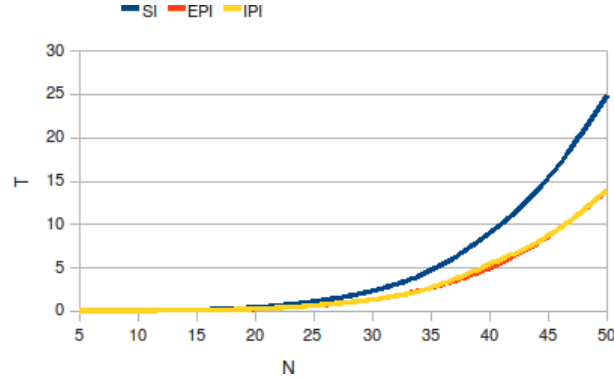
Figure 10.3: Illustration of E_s derived for EPI and IPI on DCP in relation to G_c .



10.1.2 G_s

The $\overline{\mu_s}$ derived for SI, IPI, and EPI are, in Figure 10.4, illustrated. A general performance advantage for EPI and IPI against SI is shown. Further, the $\overline{\mu_s}$ for EPI and IPI are similar.

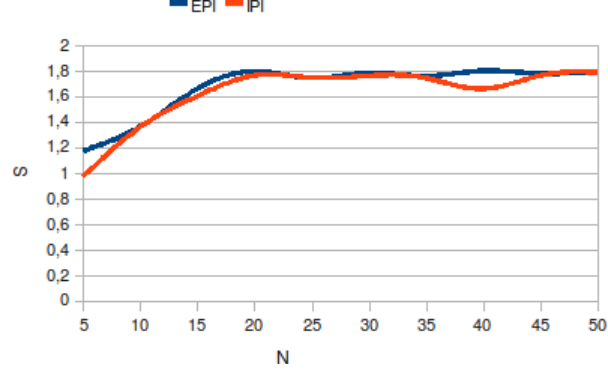
Figure 10.4: Illustration of $\overline{\mu_s}$ derived for SI, EPI, and IPI on DCP in relation to G_s .



In Figure 10.5, the derived S_s for EPI and IPI are shown. Figure 10.5 shows that S_s for EPI and IPI are similar. From $N = 5$ to $N = 20$, a significant proportional correlation between N and S is shown. This relation stagnates or becomes marginal for $N > 20$. For EPI and IPI, \overline{S} is 1.64 and 1.57, respectively. This shows a small advantage for EPI. Both EPI and IPI are algorithmically scalable, especially from $N = 5$ to $N = 20$. A small deviation from this trend can be seen for IPI where $N = 40$.

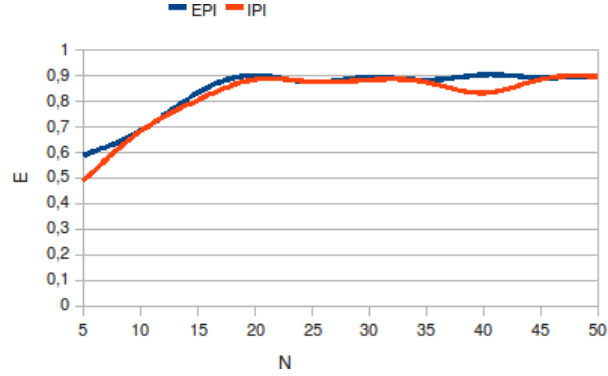
The derived E_s for EPI and IPI are shown in Figure 10.6. For EPI \overline{E} is 0.82 and for IPI \overline{E} is 0.78. δ is exceeded and IPI achieves maintenance of performance for DCP in relation to G_s . The E_s for IPI are similar to the E_s for TTE. Till

Figure 10.5: Illustration of S_s derived for EPI and IPI on DCP in relation to G_s .



$N = 20$, a proportional correlation can be seen which, hereafter, more or less stagnates. Linear speedup is not achieved for IPI. $\bar{\delta}_E$ is 0.14. Hence, there is a moderate difference between theoretical efficiency and achieved efficiency.

Figure 10.6: Illustration of E_s derived for EPI and IPI on DCP in relation to G_s .



10.1.3 Comparison

EPI and IPI achieves similar performance for both the G_c - and G_s -implementations. From an application performance viewpoint, there is only marginal performance advantage of explicit parallelization.

IPI is algorithmically scalable. IPI achieves maintenance of performance for DCP. There is a moderate difference between theoretical efficiency and achieved efficiency for IPI.

10.2 QCP

The results, analysis, and evaluation of executing the benchmark on QCP is presented. A comparison between the evaluation of the G_c - and G_s -implementations ends the section.

10.2.1 G_c

The $\bar{\mu}s$ derived for SI, IPI, and EPI are illustrated in Figure 10.7. In general, a significant performance advantage is shown for EPI and IPI against SI. The $\bar{\mu}s$ for EPI and IPI are similar with a small advantage of IPI against EPI. Still, the $\bar{\mu}s$ for EPI and IPI are similar.

Figure 10.7: Illustration of $\bar{\mu}s$ derived for SI, EPI, and IPI on QCP in relation to G_c .

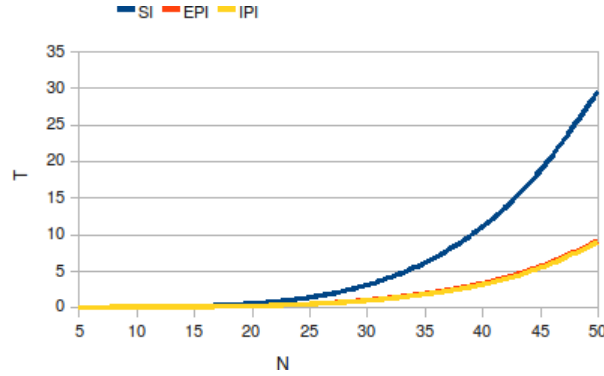


Figure 10.8 shows the Ss for EPI and IPI. For $N = 10$, there is a peculiar slope downwards and then upwards. The reason for this is not known. Due to this, let's abstract from this peculiarity through the remaining part of this section. There is an almost consistent marginal proportional correlation between N and S till $N = 40$. Hereafter, the correlation transitions to marginal invert proportional. \bar{S} for EPI is 2.61 and for IPI \bar{S} is 2.66. This shows a small advantage for IPI. Till $N = 40$, EPI and IPI are algorithmically scalable.

Figure 10.8: Illustration of Ss derived for EPI and IPI on QCP in relation to G_c .

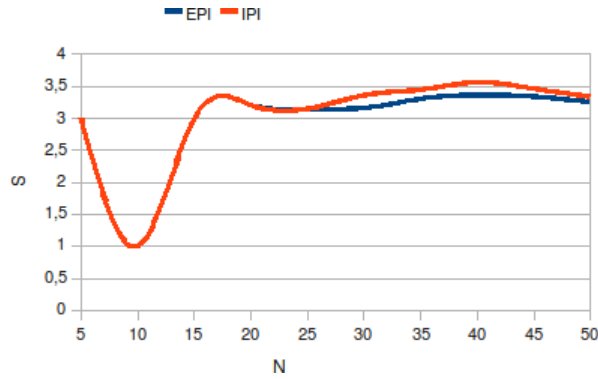
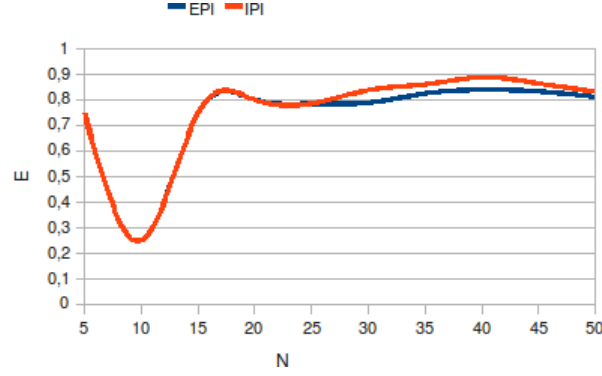


Figure 10.9 illustrates the derived Es for EPI and IPI. For EPI \bar{E} is 0.65 and for IPI \bar{E} is 0.66. δ is exceeded significantly. Hence, IPI achieves maintenance of performance for QCP in relation to G_c . There are some similarities between the Es for IPI and the Es for TTE. The initial proportional correlation for IPI is less significant than for TTE. Also, there is a marginal inverse proportional

correlation from $N = 40$ for IPI but not for TTE. For IPI, linear speedup is not achieved. $\overline{\delta_E}$ is 0.17. Therefore, the difference between theoretical efficiency and achieved efficiency is considered as moderate.

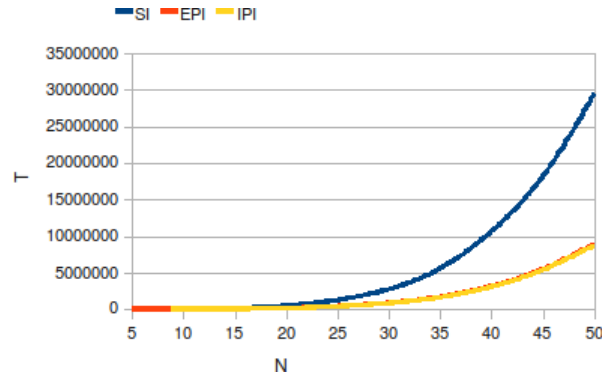
Figure 10.9: Illustration of E_s derived for EPI and IPI on QCP in relation to G_c .



10.2.2 G_s

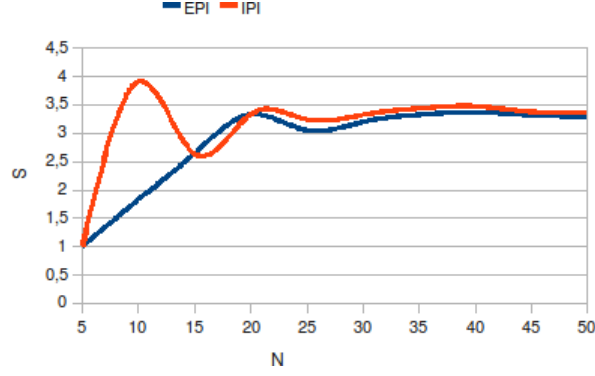
In Figure 10.10, the $\overline{\mu_s}$ derived for SI, IPI, and EPI are presented. A significant performance advantage is in general shown for EPI and IPI against SI. The $\overline{\mu_s}$ for EPI and IPI are similar. However, there is a small advantage of IPI against EPI. Despite this, $\overline{\mu_s}$ for EPI and IPI are similar.

Figure 10.10: Illustration of $\overline{\mu_s}$ derived for SI, EPI, and IPI on QCP in relation to G_s .



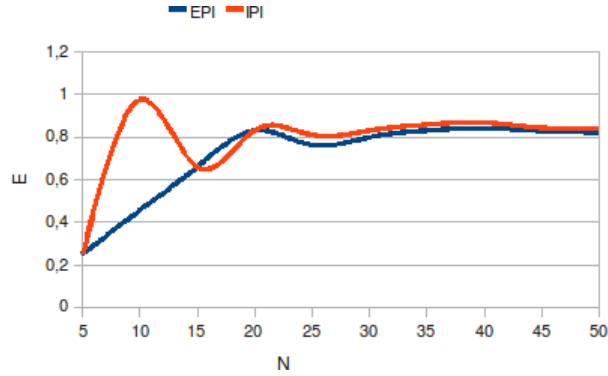
In Figure 10.5, the derived S_s for EPI and IPI are shown. For IPI and $N = 10$, there is a peculiar slope upwards and then downwards. The cause is unknown for this behavior. When not considering $N = 10$ for IPI, a significant proportional correlation from $N = 5$ to $N = 20$ is shown. This is followed by almost stagnation, specifically S varies at most by 0.27. For EPI, \overline{S} is 2.45 and for IPI \overline{S} is 2.69. Thus, a small advantage for IPI is shown. EPI and IPI are both algorithmically scalable, especially from $N = 5$ to $N = 20$.

Figure 10.11: Illustration of S_s derived for EPI and IPI on QCP in relation to G_s .



The E_s derived for EPI and IPI are shown in Figure 10.12. \overline{E}_s are 0.61 and 0.67 for EPI and IPI, respectively. This surpasses δ significantly and therefore IPI achieves maintenance of performance improvements for QCP in relation to G_s . The E_s for IPI and the E_s for TTE are similar. For both IPI and TTE, a significant proportional correlation exists from $N = 5$ to $N = 20$ followed by stagnation, more or less. Linear speedup is not achieved for IPI. $\overline{\delta_E}$ is 0.09. The difference between theoretical efficiency and achieved efficiency is considered moderate.

Figure 10.12: Illustration of E_s derived for EPI and IPI on QCP in relation to G_s .



10.2.3 Comparison

Similar performance is achieved for EPI and IPI for both the G_c - and G_s -implementations. From an application performance viewpoint, there is a small performance advantage of implicit parallelization.

IPI is algorithmically scalable. IPI achieves maintenance of performance improvements for QCP. In fact, δ is surpassed significantly. There is a moderate difference between theoretical efficiency and achieved efficiency for IPI.

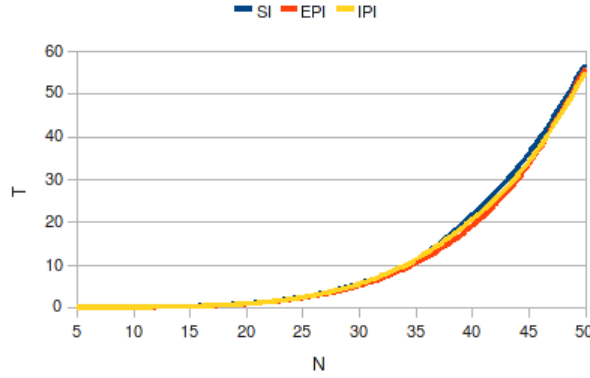
10.3 SMTP

The results, analysis, and evaluation of executing the benchmark on SMTP is here presented. In the end of section, a comparison between the evaluation of the G_c - and G_s -implementations is made.

10.3.1 G_c

In Figure 10.13, the $\bar{\mu}s$ derived for SI, IPI, and EPI are shown. The figure shows only marginal improvements of EPI and IPI over SI. $\bar{\mu}s$ for EPI and IPI are similar.

Figure 10.13: Illustration of $\bar{\mu}s$ derived for SI, EPI, and IPI on SMTP in relation to G_c .



A likely deviation is found where $N = 5$ for S_s derived for EPI and IPI. This is because $\bar{\mu}$ for $N = 5$ for SI is 2352.9 microseconds compared to 1 microsecond for both EPI and IPI. This results in $S = 2352.9$ which is clearly incorrect considering that only two processing units are used. Further, the illustration of S_s (Figure 10.14) is at best confusing. Therefore, the S_s where $N = 5$ are not considered in this section. The derived S_s , excluding where $N = 5$, are shown in Figure 10.15. Figure 10.15 shows that EPI achieves better S_s than IPI for all cases except where $N = 50$. However, the difference is within $[0.02, 0.09]$ which can be seen as marginal. Otherwise, the trends for the S_s of EPI and IPI are similar, i.e. when EPI has an upwards slope so does IPI and vice versa, except from $N = 25$ to $N = 30$. Both EPI and IPI can not be considered algorithmically scalable.

The E_s derived for EPI and IPI are shown in Figure 10.16, excluding E where $N = 5$. For EPI, \bar{E} is 0.54 and for IPI, \bar{E} is 0.52. δ is not exceeded. This means, IPI does not achieve maintenance of performance improvements for SMTP in relation to G_c . There are no significant trends shared between the E_s for IPI and the E_s for TTE. $\bar{\delta_E}$ is 0.47. Due to this, the difference between theoretical efficiency and achieved efficiency is considered significant.

Figure 10.14: Illustration of S_s derived for EPI and IPI on SMTP in relation to G_c including $N = 5$.

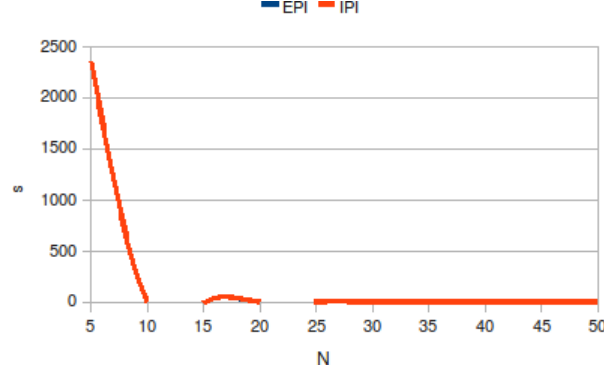
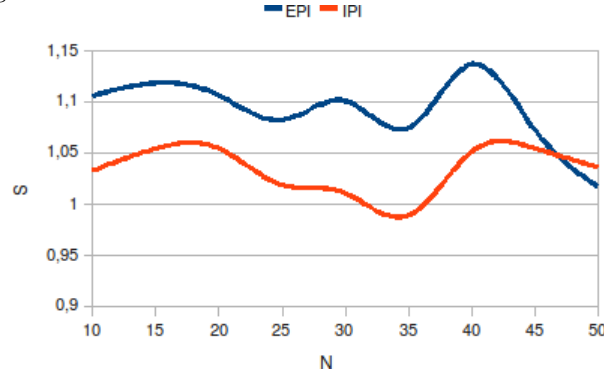


Figure 10.15: Illustration of S_s derived for EPI and IPI on SMTP in relation to G_c excluding $N = 5$.



10.3.2 G_s

The $\bar{\mu}s$ derived for SI, EPI, and IPI are shown in Figure 10.17. A marginal performance advantage is shown for EPI and IPI over SI.

In Figure 10.18, the derived S_s for EPI and IPI are illustrated. When comparing IPI to EPI, somewhat significant variances are shown for IPI from $N = 10$ to $N = 25$ where absolute leaps between distinct N s are within $[0.6, 0.7]$. Otherwise, from an abstract point of view, EPI and IPI show similar trends. Both start with an upwards slope and change at circa $N = 35$ to a downwards slope. In most cases, specifically $N = 15$, $N = 25$, and from $N = 35$ to $N = 50$, IPI show better S_s than EPI. In addition, the S_s are only arguments against EPI and IPI being algorithmically scalable.

In Figure 10.19, the E_s derived for EPI and IPI are presented. \bar{E} is 0.52 for both EPI and IPI which does not exceed δ . Therefore, IPI is not considered an adequate solution for maintenance of performance improvements for SMTP in relation to G_s . From an abstract viewpoint, there are some similarities between the E_s for IPI and the E_s for TTE. There is a proportional correlation till $N = 25$ for IPI. $\bar{\delta}_E$ is 0.45. Based on this, the difference between theoretical efficiency and achieved efficiency is considered significant.

Figure 10.16: Illustration of E_s derived for EPI and IPI on SMTP in relation to G_c excluding $N = 5$.

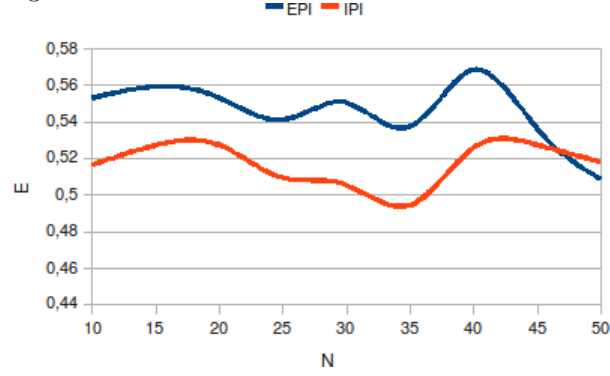
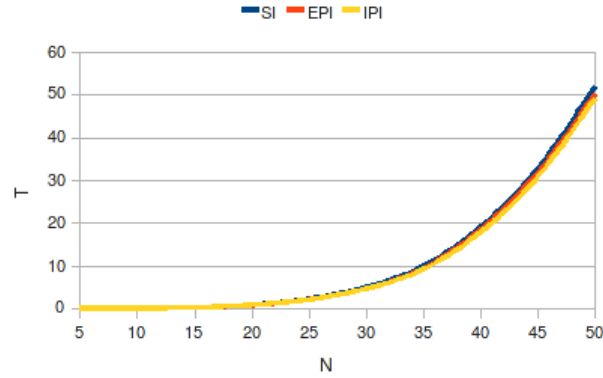


Figure 10.17: Illustration of $\bar{\mu}s$ derived for SI, EPI, and IPI on SMTP in relation to G_s .



10.3.3 Comparison

For both the G_c - and G_s -implementations, similar performance is achieved for EPI and IPI from an abstract point of view. There is no performance advantage of implicit parallelization for G_c , only an marginal disadvantage. There is a marginal performance advantage of implicit parallelization for G_s . However, all in all, almost equivalent performance achievements.

IPI is not algorithmically scalable. IPI does not achieve maintenance of performance improvements for SMTP. There is a significant difference between theoretical efficiency and achieved efficiency for IPI.

10.4 Overall Comparison

For DCP, QCP, and SMTP and for both the G_c - and G_s -implementations, similar performance is achieved for EPI and IPI.

For DCP, there is a small performance advantage of explicit parallelization. For QCP, there a small performance advantage of implicit parallelization. For SMTP, there is contextually marginal performance improvements. Considering

Figure 10.18: Illustration of S_s derived for EPI and IPI on SMTP in relation to G_s .

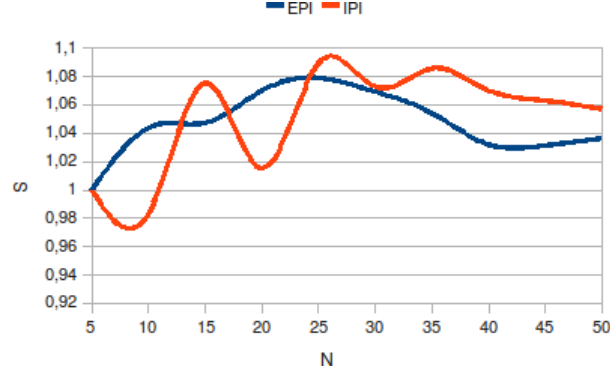
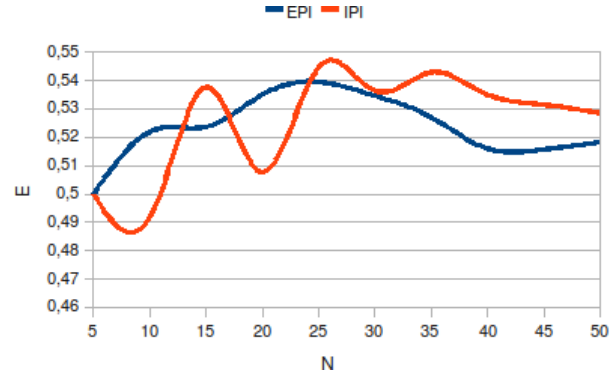


Figure 10.19: Illustration of E_s derived for EPI and IPI on SMTP in relation to G_s .



all cases, explicit- and implicit parallelization achieves, more or less, equivalent performance.

Algorithmic scalability is not achieved for IPI in relation to SMTP. For IPI in relation to both DCP and QCP, algorithmic scalability is achieved.

For DCP and IPI, \bar{E} is 0.79 for G_c and 0.78 for G_s . For QCP and IPI, \bar{E} is 0.66 for G_c and 0.67 for G_s . Thus, architectural scalability is achieved when considering DCP and IPI. For both DCP and QCD, IPI achieves maintenance of performance improvements showing an achievement of success for these platforms and this benchmark.

For DCP and QCP, there is a moderate difference between theoretical efficiency and achieved efficiency for IPI. For SMTP, there is a significant difference between theoretical efficiency and achieved efficiency for IPI.

For SMTP, the results are not successful. IPI does not achieve maintenance of performance improvements. However, the current trend of chip makers is CMPs. The maintenance of performance improvements is achieved for the platforms consisting of CMPs. When considering this fact, the results achieved can be perceived as successful.

Chapter 11

Conclusion and Future Work

The issue of the development within processor architecture has been presented. It has been argued that implicit parallelization is a potential solution. A specific solution was constructed. The solution consists of an implicitly parallel meta language called GML. GML is compiled into Erlang source with an application of an intermediate graph library called IGL. IGL and GML was constructed in relation to a graph application algorithm. This algorithm was made more explicit for the extraction of information relevant for IGL and GML. The parallelization of GML code can be summarized as follows. First, identify concurrency inherent in the GML code. Second, using the identified concurrency, extract a data flow graph consisting of threads, forks, joins, and/or parallel loops. Third, generate the target code according to the constructed data flow graph. The theoretical improvements of an implementation of the graph application algorithm in GML were significant compared to a sequential implementation.

A benchmark was designed and constructed for the purpose of testing GML. The benchmark consisted of three implementations of the graph application algorithm: a sequential, an explicitly parallel implementation in Erlang (EPI) using IGL, and finally multiple implicitly parallel implementations (IPIs) using GML. Further, the implementations were run on three different platforms: a duo core platform (DCP), a quad core platform (QCP), and an SMT platform (SMTP). The main success criteria of the benchmark was at least maintenance of performance improvements. The benchmark also concerned speedup, efficiency, and algorithmic- and architectural scalability.

The implementations for the SMTP were unsuccessful. Algorithmic scalability and maintenance of historical performance improvements were not achieved. Further, the theoretical improvements were not precise.

However, the implementations for the CMP-based platforms (DCP and QCP) showed success. The speedups and efficiencies of the IPIs were more or less equivalent to those of EPI. The theoretical improvements were somewhat precise. Algorithmic scalability and architectural scalability were achieved. Finally, and most importantly, the IPIs achieved maintenance of historical performance improvements, especially for QCP.

The major limitation of this project is that the parallelization process is designed based on working for a single graph application. Adding another or more applications for the benchmark would improve the representativeness of the benchmark results and the applicability of the parallelization approach. Perhaps, applications within different application domains could be included. This approach could on the longer term yield a more general purpose language with results supporting (or not supporting) its applicability for the tested applications.

With an execution platform consisting of many processing units, say 32 or more, the execution of the benchmark could show whether the indication of architectural scalability holds any truth. Further, such an execution platform could show whether maintenance of performance for the implicitly parallelized implementations of the benchmark of this project is consistent or simply limited to two and four processing units.

For comparison to the coarse-grained approach followed in this project, a more fine-grained solution could be interesting to see which parallelism granularity is more appropriate.

Chapter 12

Acknowledgements

Throughout the process of this project, main supervisor Daniel Ortiz-Arroyo has provided consistently good feedback. This feedback has only affected the quality of the process' product in a positive manner. Daniel also put an SMT based machine at disposal for benchmark execution.

Co-supervisor Andrea Valente contributed with good help during the initial part of this semester - especially for the decision regarding the thesis of this project.

Finally, employer Jacob Thorvald Larsen put a quad core machine at disposal for benchmark execution.

Appendix A

Extraction of Implicit Information in Algorithm 1

Below, the implicit information in Algorithm 1 is extracted to find any additional important information for IGL and GML. This results in a more elaborate algorithm. Since GML is a domain specific language, a relatively declarative algorithm is the aim. The lines not presented below are considered sufficiently explicit.

The first, seventh, and eighth line are shown successively below:

Calculate initial total entropy $H_{co_0}(G)$ and $H_{ce_0}(G)$

To solve the KPP-Pos problem, select those nodes that produce the largest change in graph entropy $H_{co_0} - H_{co_i} \geq \delta_1$

To solve the KPP-Neg problem, select those nodes that produce the largest change in graph entropy $H_{ce_0} - H_{ce_i} \geq \delta_2$

These lines show an application of variables. For instance, $H_{co_0}(G)$ in the first line is used as a variable in the eighth line. In addition, two calculations are performed in the statement: $H_{co_0}(G)$ and $H_{ce_0}(G)$. The resulting statements based on the first line are:

Line 1 : Calculate and store total connectivity entropy of G as H_{co_0}

Line 2 : Calculate and store total centrality entropy of G as H_{ce_0}

where the right hand side of **as** is the alias for the variable.

The second line states:

for all nodes \in graph G do

Within the for all loop, v_i is used implying a declaration of the variable which refer to the current nodes in the current iteration. The resulting statement is:

Line 3 : for all nodes $v_i \in$ graph G do

The third line states:

Remove node v_i , creating a modified graph G'

The statement is embedded into a for all loop. To avoid creating a loop carried dependency here, G must not be manipulated between iterations of the loop. Therefore, a replicate of G is created, stored, and manipulated as G' . This eliminates the need for the fifth line. The resulting statements are:

Line 4 : Store a copy of G as G'

Line 5 : Remove node v_i from G'

The fourth, seventh, and eighth line are shown successively below:
 Recalculate $H_{co_i}(G')$ and $H_{ce_i}(G')$, store these results
 To solve the KPP-Pos problem, select those nodes that produce the largest change in graph entropy $H_{co_0} - H_{co_i} \geq \delta_1$
 To solve the KPP-Neg problem, select those nodes that produce the largest change in graph entropy $H_{ce_0} - H_{ce_i} \geq \delta_2$
 G most likely contains multiple vertices. Hence, most likely multiple calculations of $H_{co_i}(G')$ and $H_{ce_i}(G')$ are made and stored in the fourth line. This implies the use of a variable referring to a collection. Additionally, the seventh and eighth line show an association between each v_i , H_{co_i} , and H_{ce_i} and implies that it is this association which is stored in the fourth line. To model the collections of associations, tables of fields and rows are used. The resulting statements based on the fourth line are:
 before for all:
 Line 6 : Store table with fields $\langle v, H_{co}, H_{ce} \rangle$ as H_{all}
 within for all:
 Line 7 : Add row $\langle v_i, H_{co_i}(G'), H_{ce_i}(G') \rangle$ to H_{all}

The seventh and eighth line state:
 To solve the KPP-Pos problem, select those nodes that produce the largest change in graph entropy $H_{co_0} - H_{co_i} \geq \delta_1$
 To solve the KPP-Neg problem, select those nodes that produce the largest change in graph entropy $H_{ce_0} - H_{ce_i} \geq \delta_2$
 The above lines each contains multiple embedded steps. Abstractly, equivalent sequences of steps are embedded. Solving the first solves the second, abstractly. Lets present the steps outside in. First, only v_i of H_{all} are selected. Second, at most δ_i tuples are selected. Third, the tuples of H_{all} are sorted by the nodes that produce the largest change in the specific graph entropy. The nodes which produce the greatest change are the nodes where the graph entropy after removal is the smallest. Thus, the nodes should be sorted ascending by H_{co_i} . This eliminates the need for H_{co_0} and H_{ce_0} due to removal of the application of these variables. Therefore, Line 1 and 2 are excluded in the explicit algorithm. Furthermore, δ_1 and δ_2 needs to be stored somewhere. This results in the following statements:
 Line 8 : Store δ_1 as 3 Line 9 : Store δ_2 as 3 Line 10 : Select v of the first δ_1 rows of H_{all} sorted ascending by H_{co}
 Line 11 : Select v of the first δ_2 rows of H_{all} sorted ascending by H_{ce}

All of the above transformation yields Algorithm 2.

Appendix B

Mapping Algorithm 2 to IGL APIs

Each GML STORE operation is seen as a responsibility of GML. Lets examine each step of the algorithm.

Some of the statements in Algorithm 2 use G , specifically the lines 1, 4, 5, 6, and 7. This implies an API for construction graphs. There is a need for both construction of complete graphs ($graph_complete(N)$) and sparse graphs($graph_sparse(N)$).

The first lines states:

Store table with fields $\langle v, H_{co}, H_{ce} \rangle$ as H_{all}

Here, the construction of tables are implied ($table_new()$).

The fourth line states:

for all nodes $v_i \in \text{graph } G$

This implies a need for an API to retrieve the nodes of a graph ($graph_nodes(G)$).

The fifth line states:

Store a copy of G as G'

The above implies the need for an API to copy graphs ($graph_copy(G)$).

The fixth line states:

Remove node v_i from G'

This line implies the need for an API to remove nodes in a graph ($graph_nodes_remove(G, V)$).

The seventh line states:

Add row $\langle v_i, H_{co_i}(G'), H_{ce_i}(G') \rangle$ to H_{all}

This implies, outside in, the need for an API to add rows to a table ($table_rows_add(T, R)$), an API to construct rows ($table_row_new(Cs)$), and APIs for calculating $H_{co_i}(G')$ ($graph_connectivity(G')$) and $H_{ce_i}(G')$ ($graph_centrality(G')$).

The ninth and tenth line state, respectively:

Select v of the first δ_1 tuples of H_{all} sorted ascending by H_{co}

Select v of the first δ_2 tuples of H_{all} sorted ascending by H_{ce}

These lines imply, outside in, the need for an API to select specific fields of the rows of a table ($table_select(T, Fs)$), and API to retrieve the first n rows of a table ($table_first(T, X)$), and an API to sort the rows of a table ($table_sort(T, S)$).

Appendix C

Benchmark Implementation Outcomes

The outcomes for SI, the EPIs, and the IPIs are shown in this Appendix.

C.1 SI

The outcomes for SI and G_c are presented in Table C.1 and C.2. The outcomes for SI and G_s are presented in Table C.3 and C.4.

Table C.1: SI connectivity entropy outcomes for G_c where $\delta_1 = 3$.

N	Outcome
5	(5, 2.12, 2.00), (4, 2.12, 2.00), (3, 2.12, 2.00)
10	(9, 4.68, 4.34), (6, 4.68, 4.34), (7, 4.68, 4.34)
15	(9, 7.19, 5.61), (6, 7.19, 5.61), (7, 7.19, 5.61)
20	(17, 9.70, 6.50), (9, 9.70, 6.50), (6, 9.70, 6.50)
25	(23, 12.21, 7.17), (17, 12.21, 7.17), (9, 12.21, 7.17)
30	(23, 14.71, 7.72), (17, 14.71, 7.72), (9, 14.71, 7.72)
35	(23, 17.21, 8.17), (17, 17.21, 8.17), (9, 17.21, 8.17)
40	(23, 19.71, 8.57), (39, 19.71, 8.57), (17, 19.71, 8.57)
45	(23, 22.21, 8.92), (39, 22.21, 8.92), (17, 22.21, 8.92)
50	(23, 24.71, 9.23), (39, 24.71, 9.23), (17, 24.71, 9.23)

Table C.2: SI centrality entropy outcomes for G_c where $\delta_2 = 3$.

N	Outcome
5	(5, 2.12, 2.00), (4, 2.12, 2.00), (3, 2.12, 2.00)
10	(9, 4.68, 4.34), (6, 4.68, 4.34), (7, 4.68, 4.34)
15	(9, 7.19, 5.61), (6, 7.19, 5.61), (7, 7.19, 5.61)
20	(17, 9.70, 6.50), (9, 9.70, 6.50), (6, 9.70, 6.50)
25	(23, 12.21, 7.17), (17, 12.21, 7.17), (9, 12.21, 7.17)
30	(23, 14.71, 7.72), (17, 14.71, 7.72), (9, 14.71, 7.72)
35	(23, 17.21, 8.17), (17, 17.21, 8.17), (9, 17.21, 8.17)
40	(23, 19.71, 8.57), (39, 19.71, 8.57), (17, 19.71, 8.57)
45	(23, 22.21, 8.92), (39, 22.21, 8.92), (17, 22.21, 8.92)
50	(23, 24.71, 9.23), (39, 24.71, 9.23), (17, 24.71, 9.23)

Table C.3: SI connectivity entropy outcomes for G_s where $\delta_1 = 3$.

N	Outcome
5	(4, 1.25, 0.78), (2, 1.25, 0.78), (3, 1.50, 2.00)
10	(8, 3.68, 4.50), (3, 3.68, 4.42), (7, 3.74, 4.33)
15	(4, 5.92, 6.55), (12, 5.92, 6.60), (11, 5.96, 6.50)
20	(16, 8.14, 7.99), (5, 8.14, 8.02), (15, 8.17, 8.00)
25	(20, 10.36, 9.07), (6, 10.36, 9.08), (7, 10.38, 9.05)
30	(7, 12.57, 9.93), (24, 12.57, 9.88), (23, 12.59, 9.92)
35	(8, 14.78, 10.66), (28, 14.78, 10.65), (9, 14.80, 10.62)
40	(9, 16.99, 11.30), (32, 16.99, 11.30), (31, 17.01, 11.29)
45	(36, 19.20, 11.87), (10, 19.20, 11.87), (11, 19.21, 11.87)
50	(40, 21.41, 12.31), (11, 21.41, 12.31), (39, 21.42, 12.32)

Table C.4: SI centrality entropy outcomes for G_s where $\delta_2 = 3$.

N	Outcome
5	(4, 1.25, 0.78), (2, 1.25, 0.78), (5, 1.75, 1.44)
10	(6, 3.77, 4.23), (5, 3.77, 4.23), (7, 3.74, 4.33)
15	(1, 6.19, 6.45), (7, 6.00, 6.47), (2, 6.08, 6.49)
20	(1, 8.40, 7.90), (20, 8.40, 7.91), (18, 8.26, 7.95)
25	(1, 10.61, 8.94), (25, 10.61, 8.94), (2, 10.55, 8.99)
30	(1, 12.82, 9.79), (30, 12.82, 9.80), (28, 12.72, 9.83)
35	(1, 15.02, 10.55), (2, 14.98, 10.55), (3, 14.94, 10.56)
40	(1, 17.23, 11.20), (40, 17.23, 11.21), (2, 17.19, 11.21)
45	(1, 19.44, 11.78), (45, 19.44, 11.79), (2, 19.40, 11.79)
50	(1, 21.64, 12.22), (50, 21.64, 12.23), (2, 21.61, 12.24)

C.2 EPI

The outcomes for EPI and G_c where $p = 2$ are presented in Tables C.5 and C.6. The outcomes for EPI and G_s are presented in Table C.7 and C.8.

Table C.5: EPI connectivity entropy outcomes for G_c where $\delta_1 = 3$ and $p = 2$.

N	Outcome
5	(5, 2.12, 2.00), (4, 2.12, 2.00), (3, 2.12, 2.00)
10	(8, 4.68, 4.34), (3, 4.68, 4.34), (2, 4.68, 4.34)
15	(8, 7.19, 5.61), (3, 7.19, 5.61), (2, 7.19, 5.61)
20	(17, 9.70, 6.50), (9, 9.70, 6.50), (6, 9.70, 6.50)
25	(12, 12.21, 7.17), (8, 12.21, 7.17), (20, 12.21, 7.17)
30	(22, 14.71, 7.72), (29, 14.71, 7.72), (19, 14.71, 7.72)
35	(29, 17.21, 8.17), (19, 17.21, 8.17), (26, 17.21, 8.17)
40	(23, 19.71, 8.57), (39, 19.71, 8.57), (17, 19.71, 8.57)
45	(29, 22.21, 8.92), (44, 22.21, 8.92), (19, 22.21, 8.92)
50	(23, 24.71, 9.23), (39, 24.71, 9.23), (17, 24.71, 9.23)

Table C.6: EPI centrality entropy outcomes for G_c where $\delta_2 = 3$ and $p = 2$.

N	Outcome
5	(5, 2.12, 2.00), (4, 2.12, 2.00), (3, 2.12, 2.00)
10	(8, 4.68, 4.34), (3, 4.68, 4.34), (2, 4.68, 4.34)
15	(8, 7.19, 5.61), (3, 7.19, 5.61), (2, 7.19, 5.61)
20	(17, 9.70, 6.50), (9, 9.70, 6.50), (6, 9.70, 6.50)
25	(12, 12.21, 7.17), (8, 12.21, 7.17), (20, 12.21, 7.17)
30	(22, 14.71, 7.72), (29, 14.71, 7.72), (19, 14.71, 7.72)
35	(29, 17.21, 8.17), (19, 17.21, 8.17), (26, 17.21, 8.17)
40	(23, 19.71, 8.57), (39, 19.71, 8.57), (17, 19.71, 8.57)
45	(29, 22.21, 8.92), (44, 22.21, 8.92), (19, 22.21, 8.92)
50	(23, 24.71, 9.23), (39, 24.71, 9.23), (17, 24.71, 9.23)

Table C.7: EPI connectivity entropy outcomes for G_s where $\delta_1 = 3$ and $p = 2$.

N	Outcome
5	(2, 1.25, 0.78), (4, 1.25, 0.78), (3, 1.50, 2.00)
10	(8, 3.68, 4.48), (3, 3.68, 4.42), (7, 3.74, 4.33)
15	(4, 5.92, 6.55), (12, 5.92, 6.60), (11, 5.96, 6.50)
20	(16, 8.14, 7.99), (5, 8.14, 8.02), (15, 8.17, 8.00)
25	(20, 10.36, 9.07), (6, 10.36, 9.08), (19, 10.38, 9.07)
30	(7, 12.57, 9.93), (24, 12.57, 9.88), (23, 12.59, 9.92)
35	(28, 14.78, 10.65), (8, 14.78, 10.66), (27, 14.80, 10.63)
40	(9, 16.99, 11.30), (32, 16.99, 11.30), (31, 17.01, 11.29)
45	(36, 19.20, 11.87), (10, 19.20, 11.87), (11, 19.21, 11.87)
50	(11, 21.41, 12.31), (40, 21.41, 12.31), (39, 21.42, 12.33)

Table C.8: EPI centrality entropy outcomes for G_s where $\delta_1 = 3$ and $p = 2$.

N	Outcome
5	(2, 1.25, 0.78), (4, 1.25, 0.78), (1, 1.75, 1.44)
10	(6, 3.78, 4.23), (5, 3.77, 4.23), (7, 3.74, 4.33)
15	(1, 6.19, 6.45), (7, 6.00, 6.47), (2, 6.08, 6.49)
20	(1, 8.40, 7.90), (20, 8.40, 7.91), (18, 8.26, 7.95)
25	(1, 10.61, 8.94), (25, 10.61, 8.94), (2, 10.55, 8.99)
30	(1, 12.82, 9.79), (30, 12.82, 9.80), (28, 12.72, 9.83)
35	(1, 15.02, 10.55), (2, 14.98, 10.55), (3, 14.94, 10.56)
40	(1, 17.23, 11.20), (40, 17.23, 11.21), (2, 17.19, 11.21)
45	(1, 19.44, 11.78), (45, 19.44, 11.79), (2, 19.40, 11.79)
50	(1, 21.64, 12.22), (50, 21.64, 12.23), (2, 21.61, 12.24)

The outcomes for EPI and G_c where $p = 4$ are presented in Table C.9 and C.10. The outcomes for EPI and G_s are presented in Tables C.11 and C.12.

Table C.9: EPI connectivity entropy outcomes for G_c where $\delta_1 = 3$ and $p = 4$.

N	Outcome
5	(3, 2.12, 2.00), (5, 2.12, 2.00), (4, 2.12, 2.00)
10	(8, 4.68, 4.34), (3, 4.68, 4.34), (9, 4.68, 4.34)
15	(9, 7.19, 5.61), (6, 7.19, 5.61), (7, 7.19, 5.61)
20	(20, 9.70, 6.50), (19, 9.70, 6.50), (3, 9.70, 6.50)
25	(23, 12.21, 7.17), (17, 12.21, 7.17), (9, 12.21, 7.17)
30	(11, 14.71, 7.72), (15, 14.71, 7.72), (30, 14.71, 7.72)
35	(15, 17.21, 8.17), (30, 17.21, 8.17), (35, 17.21, 8.17)
40	(23, 19.71, 8.57), (39, 19.71, 8.57), (17, 19.71, 8.57)
45	(23, 22.21, 8.92), (39, 22.21, 8.92), (17, 22.21, 8.92)
50	(11, 24.71, 9.23), (15, 24.71, 9.23), (30, 24.71, 9.23)

Table C.10: EPI centrality entropy outcomes for G_c where $\delta_1 = 3$ and $p = 4$.

N	Outcome
5	(3, 2.12, 2.00), (5, 2.12, 2.00), (4, 2.12, 2.00)
10	(8, 4.68, 4.34), (3, 4.68, 4.34), (9, 4.68, 4.34)
15	(9, 7.20, 5.61), (6, 7.20, 5.61), (7, 7.20, 5.61)
20	(20, 9.70, 6.50), (19, 9.70, 6.50), (3, 9.70, 6.50)
25	(23, 12.21, 7.17), (17, 12.21, 7.17), (9, 12.21, 7.17)
30	(11, 14.71, 7.72), (15, 14.71, 7.72), (30, 14.71, 7.72)
35	(15, 17.21, 8.17), (30, 17.21, 8.17), (35, 17.21, 8.17)
40	(23, 19.71, 8.57), (39, 19.71, 8.57), (17, 19.71, 8.57)
45	(23, 22.21, 8.92), (39, 22.21, 8.92), (17, 22.21, 8.92)
50	(11, 24.71, 9.23), (15, 24.71, 9.23), (30, 24.71, 9.23)

Table C.11: EPI connectivity entropy outcomes for G_s where $\delta_1 = 3$ and $p = 4$.

N	Outcome
5	(4, 1.25, 0.78), (2, 1.25, 0.78), (3, 1.50, 2.00)
10	(8, 3.68, 4.48), (3, 3.68, 4.42), (7, 3.74, 4.33)
15	(4, 5.92, 6.55), (12, 5.92, 6.60), (11, 5.96, 6.50)
20	(16, 8.14, 7.99), (5, 8.14, 8.02), (15, 8.17, 8.00)
25	(20, 10.36, 9.07), (6, 10.36, 9.08), (7, 10.38, 9.05)
30	(7, 12.57, 9.93), (24, 12.57, 9.88), (8, 12.59, 9.96)
35	(28, 14.78, 10.65), (8, 14.78, 10.66), (27, 14.80, 10.63)
40	(9, 16.99, 11.30), (32, 16.99, 11.30), (10, 17.01, 11.29)
45	(10, 19.20, 11.87), (36, 19.20, 11.87), (35, 19.21, 11.86)
50	(11, 21.41, 12.31), (40, 21.41, 12.31), (12, 21.42, 12.33)

Table C.12: EPI centrality entropy outcomes for G_s where $\delta_1 = 3$ and $p = 4$.

N	Outcome
5	(4, 1.25, 0.78), (2, 1.25, 0.78), (5, 1.75, 1.44)
10	(5, 3.77, 4.23), (6, 3.77, 4.23), (7, 3.74, 4.33)
15	(1, 6.19, 6.45), (7, 6.00, 6.47), (2, 6.08, 6.49)
20	(1, 8.40, 7.90), (20, 8.40, 7.91), (18, 8.26, 7.95)
25	(1, 10.61, 8.94), (25, 10.61, 8.94), (2, 10.55, 8.99)
30	(1, 12.82, 9.79), (30, 12.82, 9.80), (28, 12.72, 9.83)
35	(1, 15.02, 10.55), (2, 14.98, 10.55), (3, 14.94, 10.56)
40	(1, 17.23, 11.20), (40, 17.23, 11.21), (2, 17.19, 11.21)
45	(1, 19.44, 11.78), (45, 19.44, 11.79), (2, 19.40, 11.79)
50	(1, 21.64, 12.22), (50, 21.64, 12.23), (2, 21.61, 12.24)

C.3 IPI outcomes

The outcomes for IPI and G_c where $p = 2$ are presented in Table C.13 and C.14. The outcomes for IPI and G_s are presented in Table C.15 and C.16.

Table C.13: IPI connectivity entropy outcomes for G_c where $\delta_1 = 3$ and $p = 2$.

N	Outcome
5	(3, 2.12, 2.00), (2, 2.12, 2.00), (1, 2.12, 2.00)
10	(8, 4.68, 4.34), (3, 4.68, 4.34), (2, 4.68, 4.34)
15	(9, 7.19, 5.61), (6, 7.19, 5.61), (7, 7.19, 5.61)
20	(17, 9.70, 6.50), (9, 9.70, 6.50), (6, 9.70, 6.50)
25	(12, 12.21, 7.17), (8, 12.21, 7.17), (20, 12.21, 7.17)
30	(22, 14.71, 7.72), (29, 14.71, 7.72), (19, 14.71, 7.72)
35	(29, 17.21, 8.17), (19, 17.21, 8.17), (26, 17.21, 8.17)
40	(23, 19.71, 8.57), (39, 19.71, 8.57), (17, 19.71, 8.57)
45	(29, 22.21, 8.92), (44, 22.21, 8.92), (19, 22.21, 8.92)
50	(23, 24.71, 9.23), (39, 24.71, 9.23), (17, 24.71, 9.23)

Table C.14: IPI centrality entropy outcomes for G_c where $\delta_1 = 3$ and $p = 2$.

N	Outcome
5	(3, 2.12, 2.00), (2, 2.12, 2.00), (1, 2.12, 2.00)
10	(8, 4.68, 4.34), (3, 4.68, 4.34), (2, 4.68, 4.34)
15	(9, 7.19, 5.61), (6, 7.19, 5.61), (7, 7.19, 5.61)
20	(17, 9.70, 6.50), (9, 9.70, 6.50), (6, 9.70, 6.50)
25	(12, 12.21, 7.17), (8, 12.21, 7.17), (20, 12.21, 7.17)
30	(22, 14.71, 7.72), (29, 14.71, 7.72), (19, 14.71, 7.72)
35	(29, 17.21, 8.17), (19, 17.21, 8.17), (26, 17.21, 8.17)
40	(23, 19.71, 8.57), (39, 19.71, 8.57), (17, 19.71, 8.57)
45	(29, 22.21, 8.92), (44, 22.21, 8.92), (19, 22.21, 8.92)
50	(23, 24.71, 9.23), (39, 24.71, 9.23), (17, 24.71, 9.23)

Table C.15: IPI connectivity entropy outcomes for G_s where $\delta_1 = 3$ and $p = 2$.

N	Outcome
5	(2, 1.25, 0.78), (4, 1.25, 0.78), (3, 1.50, 2.00)
10	(8, 3.68, 4.50), (3, 3.68, 4.42), (7, 3.74, 4.33)
15	(4, 5.92, 6.55), (12, 5.92, 6.60), (11, 5.96, 6.50)
20	(16, 8.14, 7.99), (5, 8.14, 8.02), (15, 8.17, 8.00)
25	(20, 10.36, 9.07), (6, 10.36, 9.08), (19, 10.38, 9.05)
30	(7, 12.57, 9.93), (24, 12.57, 9.88), (23, 12.59, 9.92)
35	(28, 14.78, 10.66), (8, 14.78, 10.65), (27, 14.80, 10.62)
40	(9, 16.99, 11.30), (32, 16.99, 11.30), (31, 17.01, 11.29)
45	(36, 19.20, 11.87), (10, 19.20, 11.87), (11, 19.21, 11.87)
50	(11, 21.41, 12.31), (40, 21.41, 12.31), (39, 21.42, 12.32)

Table C.16: IPI centrality entropy outcomes for G_s where $\delta_1 = 3$ and $p = 2$.

N	Outcome
5	(2, 1.25, 0.78), (4, 1.25, 0.78), (1, 1.75, 1.44)
10	(6, 3.77, 4.23), (5, 3.77, 4.23), (7, 3.74, 4.33)
15	(1, 6.19, 6.45), (7, 6.00, 6.47), (2, 6.08, 6.49)
20	(1, 8.40, 7.90), (20, 8.40, 7.91), (18, 8.26, 7.95)
25	(1, 10.61, 8.94), (25, 10.61, 8.94), (2, 10.55, 8.99)
30	(1, 12.82, 9.79), (30, 12.82, 9.80), (28, 12.72, 9.83)
35	(1, 15.02, 10.55), (2, 14.98, 10.55), (3, 14.94, 10.56)
40	(1, 17.23, 11.20), (40, 17.23, 11.21), (2, 17.19, 11.21)
45	(1, 19.44, 11.78), (45, 19.44, 11.79), (2, 19.40, 11.79)
50	(1, 21.64, 12.22), (50, 21.64, 12.23), (2, 21.61, 12.24)

The outcomes for IPI and G_c where $p = 4$ are presented in Table C.17 and C.18. The outcomes for IPI and G_s are presented in Table C.19 and C.20.

Table C.17: EPI connectivity entropy outcomes for G_c where $\delta_1 = 3$ and $p = 4$.

N	Outcome
5	(5, 2.12, 2.00), (4, 2.12, 2.00), (3, 2.12, 2.00)
10	(2, 4.68, 4.34), (10, 4.68, 4.34), (1, 4.68, 4.34)
15	(14, 7.19, 5.61), (4, 7.19, 5.61), (12, 7.19, 5.61)
20	(17, 9.70, 6.50), (9, 9.70, 6.50), (6, 9.70, 6.50)
25	(23, 12.21, 7.17), (17, 12.21, 7.17), (9, 12.21, 7.17)
30	(11, 14.71, 7.72), (15, 14.71, 7.72), (30, 14.71, 7.72)
35	(15, 17.21, 8.17), (30, 17.21, 8.17), (35, 17.21, 8.17)
40	(23, 19.71, 8.57), (39, 19.71, 8.57), (17, 19.71, 8.57)
45	(23, 22.21, 8.92), (39, 22.21, 8.92), (17, 22.21, 8.92)
50	(11, 24.71, 9.23), (15, 24.71, 9.23), (30, 24.71, 9.23)

Table C.18: EPI centrality entropy outcomes for G_c where $\delta_1 = 3$ and $p = 4$.

N	Outcome
5	(5, 2.12, 2.00), (4, 2.12, 2.00), (3, 2.12, 2.00)
10	(2, 4.68, 4.34), (10, 4.68, 4.34), (1, 4.68, 4.34)
15	(14, 7.19, 5.61), (4, 7.19, 5.61), (12, 7.19, 5.61)
20	(17, 9.70, 6.50), (9, 9.70, 6.50), (6, 9.70, 6.50)
25	(23, 12.21, 7.17), (17, 12.21, 7.17), (9, 12.21, 7.17)
30	(11, 14.71, 7.72), (15, 14.71, 7.72), (30, 14.71, 7.72)
35	(15, 17.21, 8.17), (30, 17.21, 8.17), (35, 17.21, 8.17)
40	(23, 19.71, 8.57), (39, 19.71, 8.57), (17, 19.71, 8.57)
45	(23, 22.21, 8.92), (39, 22.21, 8.92), (17, 22.21, 8.92)
50	(11, 24.71, 9.23), (15, 24.71, 9.23), (30, 24.71, 9.23)

Table C.19: EPI connectivity entropy outcomes for G_s where $\delta_1 = 3$ and $p = 4$.

N	Outcome
5	(4, 1.25, 0.78), (2, 1.25, 0.78), (3, 1.50, 2.00)
10	(8, 3.68, 4.50), (3, 3.68, 4.42), (7, 3.74, 4.33)
15	(4, 5.92, 6.55), (12, 5.92, 6.60), (11, 5.96, 6.50)
20	(16, 8.14, 7.99), (5, 8.14, 8.02), (15, 8.17, 8.00)
25	(20, 10.36, 9.07), (6, 10.36, 9.08), (7, 10.38, 9.05)
30	(7, 12.57, 9.93), (24, 12.57, 9.88), (8, 12.59, 9.92)
35	(8, 14.78, 10.66), (28, 14.78, 10.65), (27, 14.80, 10.62)
40	(9, 16.99, 11.30), (32, 16.99, 11.30), (10, 17.01, 11.29)
45	(10, 19.20, 11.87), (36, 19.20, 11.87), (35, 19.21, 11.87)
50	(11, 21.41, 12.31), (40, 21.41, 12.31), (12, 21.42, 12.32)

Table C.20: EPI centrality entropy outcomes for G_s where $\delta_1 = 3$ and $p = 4$.

N	Outcome
5	(4, 1.25, 0.78), (2, 1.25, 0.78), (5, 1.75, 1.44)
10	(5, 3.77, 4.23), (6, 3.77, 4.23), (7, 3.74, 4.33)
15	(1, 6.19, 6.45), (7, 6.00, 6.47), (2, 6.08, 6.49)
20	(1, 8.40, 7.90), (20, 8.40, 7.91), (18, 8.26, 7.95)
25	(1, 10.61, 8.94), (25, 10.61, 8.94), (2, 10.55, 8.99)
30	(1, 12.82, 9.79), (30, 12.82, 9.80), (28, 12.72, 9.83)
35	(1, 15.02, 10.55), (2, 14.98, 10.55), (3, 14.94, 10.56)
40	(1, 17.23, 11.20), (40, 17.23, 11.21), (2, 17.19, 11.21)
45	(1, 19.44, 11.78), (45, 19.44, 11.79), (2, 19.40, 11.79)
50	(1, 21.64, 12.22), (50, 21.64, 12.23), (2, 21.61, 12.24)

Bibliography

- [1] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: the spi calculus. In *CCS '97: Proceedings of the 4th ACM conference on Computer and communications security*, pages 36–47, New York, NY, USA, 1997. ACM.
- [2] Adeel Abbas and Affan Ahmad. Object oriented parallel programming. volume 1, pages 89–93, College of E & ME, NUST Pakistan, 2002. IEEE.
- [3] Ferri Abolhassan, Reinhard Drefenstedt, Jörg Keller, Wolfgang J. Paul, and Dieter Scheerer. On the physical design of prams. 1993.
- [4] Anant Agarwal and Markus Levy. The kill rule for multicore. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 750–753, New York, NY, USA, 2007. ACM.
- [5] Bowen Alpern, Larry Carter, and Jeanne Ferrante. Modeling parallel computers as memory hierarchies. In *Programming Models for Massively Parallel Computers*, pages 116–123. IEEE Computer Society Press, 1993.
- [6] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. pages 79–81, 2000.
- [7] James H. Anderson, John M. Calandrino, and UmaMaheswari C. Devi. Real-time scheduling on multicore platforms. In *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 179–190, Washington, DC, USA, 2006. IEEE Computer Society.
- [8] ANTLR v3. WWW page, 2009. <http://www.antlr.org/>.
- [9] ANTLRWorks: The ANTLR GUI Development Environment. WWW page, 2009. <http://www.antlr.org/works/index.html>.
- [10] Jorge L. Ortega Arjona and Graham Roberts. Architectural patterns for parallel programming. April 1998.
- [11] Joe Armstrong. Concurrency oriented programming in erlang. 2003.
- [12] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

- [13] 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. Multi-core programming approach in the real-time virtual instrumentation. *Instrumentation and Measurement Technology Conference Proceedings*, December 2006.
- [14] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38:1526–1538, 1989.
- [15] Guy E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, 1996.
- [16] W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the perfect benchmarks programs. *IEEE Trans. Parallel Distrib. Syst.*, 3(6):643–656, 1992.
- [17] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Control flow analysis for the pi-calculus. In *CONCUR '98: Proceedings of the 9th International Conference on Concurrency Theory*, pages 84–98, London, UK, 1998. Springer-Verlag.
- [18] Matthew Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin, and David August. Revisiting the sequential programming model for multi-core. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 69–84, Washington, DC, USA, 2007. IEEE Computer Society.
- [19] U. Bruening, Wolfgang K. Giloi, and Wolfgang Schroeder-Preikschat. Latency hiding in message-passing architectures. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 704–709, Washington, DC, USA, 1994. IEEE Computer Society.
- [20] Mikkel Bundgaard and Vladimiro Sassone. Typed polyadic pi-calculus in bigraphs. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 1–12, New York, NY, USA, 2006. ACM.
- [21] Marco Carbone, Mogens Nielsen, and Vladimiro Sassone. A calculus for trust management. In *In Proceedings from Foundations of Software Technology and Theoretical Computer Science: 24th International Conference (FSTTCS04)*, pages 161–173. Springer, 2004.
- [22] Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. Types as models: model checking message-passing programs. *SIGPLAN Not.*, 37(1):45–57, 2002.
- [23] Satish Chandra, James R. Larus, and Anne Rogers. Where is time spent in message-passing and shared-memory programs? *SIGPLAN Not.*, 29(11):61–73, 1994.
- [24] Ding-Kai Chen, Hong-Men Su, and Pen-Chung Yew. The impact of synchronization and granularity on parallel systems. *SIGARCH Comput. Archit. News*, 18(3a):239–248, 1990.

- [25] Silvia Crafa and Sabina Rossi. P-congruences as non-interference for the pi-calculus. In *FMSE '06: Proceedings of the fourth ACM workshop on Formal methods in security*, pages 13–22, New York, NY, USA, 2006. ACM.
- [26] Ct Technology: a new perspective on Data-parallel Programming. WWW page, 2009. <http://software.intel.com/en-us/data-parallel/>.
- [27] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: towards a realistic model of parallel computation. In *PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12, New York, NY, USA, 1993. ACM.
- [28] David E. Culler, Richard M. Karp, David Patterson, Abhijit Sahay, Eunice E. Santos, Klaus Erik Schauser, Ramesh Subramonian, and Thorsten von Eicken. Logp: a practical model of parallel computation. *Commun. ACM*, 39(11):78–85, 1996.
- [29] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.
- [30] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. Simple, efficient shared memory simulations. In *SPAA '93: Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 110–119, New York, NY, USA, 1993. ACM.
- [31] Charles Donnelly and Richard Stallman. Bison - the yacc-compatible parser generator. 2009.
- [32] Kevin Dowd and Charles Severance. *High Performance Computing*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998.
- [33] Chris Edwards. The many lives of moore's law. *Engineering & Technology*, 3(1):36–39, 2008.
- [34] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, 1997.
- [35] Jakob Ehmsen. Analysis and Evaluation of Modern Language Support for Programming Multicore Processors. Master's thesis, Esbjerg Institute of Technology - Aalborg University, January 2009.
- [36] Eric Allen Engle. Extraterritorial jurisdiction: Can rico protect human rights? a computer analysis of a semi-determinate legal question. *Suffolk University Journal of High Technology Law*, 3(1):1–28, 2004.
- [37] Erlang digraph module online manual. WWW page, 2009. <http://erlang.org/doc/man/digraph.html>.
- [38] Erlang timer module online manual. WWW page, 2009. <http://erlang.org/doc/man/timer.html>.

- [39] Faith E. Fich, Prabhakar L. Ragde, and Avi Wigderson. Relations between concurrent-write models of parallel computation. In *PODC '84: Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 179–189, New York, NY, USA, 1984. ACM.
- [40] Steven Fortune and James Wyllie. Parallelism in random access machines. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118, New York, NY, USA, 1978. ACM.
- [41] Vincent W. Freeh and Vincent W. Freeh. A comparison of implicit and explicit parallel programming. Technical report, University of Arizona, 1994.
- [42] Anwar Ghuloum, Terry Smith, Gansha Wu, Xin Zhou, Jesse Fang, Peng Guo, Byoungro So, Mohan Rajagopalan, Yongjian Chen, and Biao Chen. Future-proof data parallel algorithms and software on intel®multi-core architecture. *Intel Technology Journal*, 2007.
- [43] P. B. Gibbons. A more practical pram model. In *SPAA '89: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 158–168, New York, NY, USA, 1989. ACM.
- [44] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The javaTMlanguage specification - third edition. Technical report.
- [45] Graph representation. WWW page, 2009. <http://www.cs.toronto.edu/~heap/270F02/node35.html>.
- [46] Douglas Gregor and Andrew Lumsdaine. The parallel bgl: A generic library for distributed graph computations. In *In Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.
- [47] John L. Gustafson. Reevaluating amdahl’s law. *Commun. ACM*, 31(5):532–533, 1988.
- [48] A. K. Hartmann and M. Weigt. Introduction to graphs. *ArXiv Condensed Matter e-prints*, February 2006.
- [49] Oltea Mihaela Herescu. *The probabilistic asynchronous pi-calculus*. PhD thesis, 2002. Adviser-Catuscia Palamidessi.
- [50] Karem Briceño Hernández. Parallelization of scientific legacy code. Master’s thesis, Texas Tech University, 2003.
- [51] Mark D. Hill. What is scalability? *SIGARCH Comput. Archit. News*, 18(4):18–21, 1990.
- [52] Roberto Hoyos. Inside parallel computing: present and future. 2006.
- [53] Intel®CoreTMi7 Processor. WWW page, 2009. <http://www.intel.com/products/processor/corei7/index.htm>.

- [54] Wen-mei Hwu, Shane Ryoo, Sain-Zee Ueng, John H. Kelm, Isaac Gelado, Sam S. Stone, Robert E. Kidd, Sara S. Baghsorkhi, Aqeel A. Mahesri, Stephanie C. Tsao, Nacho Navarro, Steve S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Implicitly parallel programming models for thousand-core microprocessors. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 754–759, New York, NY, USA, 2007. ACM.
- [55] Revolutionizing How We Use Technology Today and Beyond. WWW page, 2008. http://www.intel.com/technology/architecture-silicon/32nm/index.htm?cid=emea:ggl|chips_dk_32nm_en|d5BCFAB|s.
- [56] Intel®Demonstrates Industry’s First 32nm Chip and Next-Generation Nehalem Microprocessor Architecture. WWW page, 2008. http://www.intel.com/pressroom/archive/releases/20070918corp_a.htm.
- [57] Introduction to Intel®Core™Duo Processor architecture. WWW page, 2009. http://www.intel.com/technology/itj/2006/volume10issue02/art01_Intro_to_Core_Duo/p01_abstract.htm.
- [58] Java Compiler Compiler™(JavaCC™) - The Java Parser Generator. WWW page, 2009. <https://javacc.dev.java.net/>.
- [59] Stephen C. Johnson. Yacc: Yet another compiler compiler. 1975.
- [60] Brian K. Justice. A Study on Parallel Processing and Related Computer Graphics. Master’s thesis, South Carolina Honors College, September 1995.
- [61] Timothy Harold Kaiser. *Dynamic load distributions for adaptive computations on MIMD machines using hybrid genetic algorithms*. PhD thesis, Albuquerque, NM, USA, 1997.
- [62] Richard M. Karp. A survey of parallel algorithms for shared-memory machines. Technical report, Berkeley, CA, USA, 1988.
- [63] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1978.
- [64] Christoph Kessler and Jörg Keller. Models for parallel computing: Review and perspectives. *Mitteilungen - Gesellschaft fr Informatik e. V., Parallel-Algorithmen und Rechnerstrukturen*, 24:?, 2007.
- [65] David Jonathan King. *Functional Programming and Graph Algorithms*. PhD thesis, 1996.
- [66] Vipin Kumar, George Karypis, and Ananth Grama. Role of message-passing in performance oriented parallel programming. In *Proceedings of the Eighth SIAM Parallel Processing Conference*, March 1997.
- [67] M. E. Lesk and E. Schmidt. Lex - a lexical analyser generator. 1975.
- [68] Kenneth C. Loudon. *Compiler Construction: Principles and Practice*. PWS Publishing Company, 1997.

- [69] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. 2007.
- [70] Kenneth Lundin. Inside the erlang vm - with focus on smp, 2008.
- [71] Tim Mattson and Michael Wrinn. Parallel programming: can we please get it right this time? In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 7–11, New York, NY, USA, 2008. ACM.
- [72] Marc Mazzariol. Computer-aided parallellization of applications. 2001.
- [73] Time and Frequency from A to Z. WWW page, 2009. <http://tf.nist.gov/general/enc-m.htm>.
- [74] Dr. Gordon E. Moore. Cramming more components into integrated circuits. *Electronics*, 38(8):111, 1965.
- [75] Dr. Gordon E. Moore. International solid-state circuits conference. San Fransisco, Calif., USA, February 2003. Intel.
- [76] 32-core CPUs from Intel and AMD. WWW page, 2009. <http://blogs.jmls.edu/ITSHDBlog/Lists/Posts/Post.aspx?List=5af27beb-9d65-429a-9ba3-0229e0530ddd&ID=28>.
- [77] Intel unveils Nehalem-EX, demos 64 core system. WWW page, 2009. <http://www.nordichardware.com/news,9350.html>.
- [78] AMD announces 6-core and 12-core Opteron processors. WWW page, 2009. <http://www.tgdaily.com/content/view/37323/135/>.
- [79] AMD pulls in six-core CPU, announces 16-core for 2011. WWW page, 2009. <http://www.tgdaily.com/content/view/42125/135/>.
- [80] Intel to detail 8-core server chip. WWW page, 2009. http://news.cnet.com/8301-13924_3-10244564-64.html.
- [81] Intel builds 80-core prototype. WWW page, 2009. http://www.bit-tech.net/news/hardware/2007/01/18/intel_builds_80-core_prototype/1.
- [82] Vincenzo Nicosia. Towards hard real-time erlang. In *ERLANG '07: Proceedings of the 2007 SIGPLAN workshop on ERLANG Workshop*, pages 29–36, New York, NY, USA, 2007. ACM.
- [83] Thomas Noll and Chanchal Kumar Roy. Modeling erlang in the pi-calculus. In *ERLANG '05: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, pages 72–77, New York, NY, USA, 2005. ACM.
- [84] Daniel Ortiz-Arroyo and D. M. Hussain. An information theory approach to identify sets of key players. In *EuroISI '08: Proceedings of the 1st European Conference on Intelligence and Security Informatics*, pages 15–26, Berlin, Heidelberg, 2008. Springer-Verlag.
- [85] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, 1986.

- [86] Vern Paxson. Flex, version 2.5. 1995.
- [87] Benjamin C. Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *J. ACM*, 47(3):531–584, 2000.
- [88] Benjamin C. Pierce and David N. Turner. Pict: a programming language based on the pi-calculus. pages 455–494, 2000.
- [89] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.
- [90] Think Parallel or Perish. WWW page, 2008. <http://www.devx.com/go-parallel/Article/32784>.
- [91] Martin C. Rinard and Monica S. Lam. The design, implementation, and evaluation of jade. *ACM Trans. Program. Lang. Syst.*, 20(3):483–545, 1998.
- [92] Robin Milner. The polyadic π -calculus: a tutorial. (ECS-LFCS-91-180), October 1991. Also in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer and H. Schwichtenberg, , 1993.
- [93] Kenneth H. Rosen. *Discrete mathematics and its applications (5th edition)*. McGraw-Hill, Inc., New York, NY, USA, 2002.
- [94] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–15, New York, NY, USA, 2008. ACM.
- [95] David B. Skillicorn. Architecture-independent parallel computation. *Computer*, 23(2):38–50, December 1990.
- [96] J. E. Smith. Characterizing computer performance with a single number. *Commun. ACM*, 31(10):1202–1206, 1988.
- [97] Limited-memory, limited-CPU environments. WWW page, 2009. <http://www.joelonsoftware.com/items/2007/09/18.html>.
- [98] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs's Journal*, 30(3), 2005.
- [99] Alexandre Tiskin. The bulk-synchronous parallel random access machine. *Theor. Comput. Sci.*, 196(1-2):109–130, 1998.
- [100] Predrag Tosic and Gul Agha. On parallel vs. sequential threshold cellular automata. 2004.
- [101] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 392–403, New York, NY, USA, 1995. ACM.

- [102] Ilkka Tuomi. The lives and the death of moore's law. October 2002.
- [103] David N. Turner. The polymorphic pi-calculus: Theory and implementation. Technical report, University of Edinburgh, 1995.
- [104] Philip Wadler. An angry half-dozen. *SIGPLAN Not.*, 33(2):25–30, 1998.
- [105] William M. Waite and Gerhard Goos. Compiler construction, 1984.
- [106] Claes Wikstrom. Distributed programming in erlang. In *In PASC0'94 - First International Symposium on Parallel Symbolic Computation*, pages 412–421, 1994.
- [107] Barry Wilkinson and Michael Allen. *Parallel programming: techniques and applications using networked workstations and parallel computers*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.
- [108] Barry Wilkinson and Michael Allen. A state-wide senior parallel programming course. *IEEE Transactions on Education*, 42(3):167–173, 1999.
- [109] Henry Xiao. Parallel graph algorithms - algorithmic graph theory study report. 2003.
- [110] Laura A. Zager. Graph similarity and matching. 2005.
- [111] Lars ke Fredlund, Dilian Gurov, Thomas Noll, Mads Dam, Thomas Arts, and Gennady Chugunov. A verification tool for erlang. *International Journal on Software Tools for Technology Transfer*, 4(4):405–420, 2003. <http://www.sics.se/~mfd/sttt.ps>.