UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
BACHELOR OF COMPUTER SCIENCE

KAUE SOARES DA SILVEIRA

A Comparative Study of Programming Models for Concurrency

Final Report presented in partial fulfillment of the requirements for the degree of Bachelor of Computer Science

Dr. Cláudio Fernando Resin Geyer
Advisor

Dr. Sebastian Nanz, ETH Zurich
Coadvisor

Porto Alegre, July 2012

CIP – CATALOGING-IN-PUBLICATION

Silveira, Kaue Soares da

A Comparative Study of Programming Models for Concurrency / Kaue Soares da Silveira. – Porto Alegre: COMGRAD CIC UFRGS, 2012.

74 f.: il.

Final Report (Bachelor) – Universidade Federal do Rio Grande do Sul. Bachelor of Computer Science, Porto Alegre, BR, 2012. Advisor: Cláudio Fernando Resin Geyer; Coadvisor: Sebastian Nanz, ETH Zurich.

1. Concurrency. 2. Usability. 3. Programming languages. I. Geyer, Cláudio Fernando Resin. II. Nanz, Sebastian.

*"Luck is what happens
when preparation meets opportunity."*
— Seneca

ACKNOWLEDGEMENTS

First, I thank my classmates on UFRGS and in the exchange program, especially Bruno Coswig Fiss, Gabriel Marques Portal and Leonardo Piletti Chatain, for promoting a healthy competition and serving as an incentive for self improvement.

Second, I thank my professors in Brazil and in France, especially Alexandre da Silva Carissimi, Alvaro Freitas Moreira, Aline Villavicencio, Marcus Rolf Petter Ritt and Nicolas Bruno Maillard, for creating challenging exams and being open for discussion about their answers.

Next, I thank my advisors Cláudio Fernando Resin Geyer and Sebastian Nanz for all the great insights and bright contributions they have done to this work.

Last, I thank my family, especially my parents Plinio Omar Silva da Silveira and Lucia da Rosa Soares, for giving me all the support through my life and for motivating me to overcome all the obstacles in my path.

Thank you all very much!

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

CPU	Central Processing Unit
CSP	Communicating Sequential Processes
HPC	High Performance Computing
IO	Input / Output
LCG	Linear Congruential Generator
LoC	Lines of Source Code
NoW	Number of Words
PGAS	Partitioned Global Address Space
PID	Process Identifier
PRAM	Parallel Random Access Machine
RAM	Random-Access Memory
SCOOP	Simple Concurrent Object Oriented Programming
TBB	Threading Building Blocks

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ABSTRACT

The recent transition of general purpose computing to multi-core architectures has drastically changed the computing field. This transition was caused by the physical constraints preventing frequency scaling. It is not the case anymore that the next generation of hardware will make the programs much faster, unless they are created with concurrency in mind – a task that has long been recognized as difficult and error-prone.

Although many programming languages have been proposed to make writing correct concurrent programs simpler than traditional multi-threading, it is largely unknown whether any of the newly created languages actually improves some aspect of programmer productivity. This work focuses on the subset of those languages that are oriented towards multi-core architectures.

The goal of this project is to provide a detailed comparison of the different approaches, discuss their strengths and weaknesses and identify the application areas they are best suited for. This required solving relevant problems (from a concurrency point of view) using each approach. It was also necessary to decide on the methodology of the comparison.

The results include: a survey of related work; a methodology for comparing concurrent programming languages; and a study showing the applicability of the methodology, where 6 languages are compared in terms of time to code, source code size, execution time and speedup through the solution of 6 problems.

Keywords: Concurrency, usability, programming languages.

Estudo Comparativo de Modelos de Programação Concorrente

RESUMO

A recente transição da computação de propósito geral para arquiteturas *multi-core* mudou drasticamente o campo da computação. Esta transição foi causada pelas limitações físicas que impedem que se continue aumentando a frequência dos processadores. Não é mais o caso que a próxima geração de hardware fará com que os programas fiquem muito mais rápidos, a menos que eles sejam criados com concorrência em mente – uma tarefa que tem sido há muito tempo considerada difícil e muito propensa a erros.

Embora muitas linguagens de programação tenham sido propostas para tornar a escrita de programas concorrentes mais fácil que o *multithreading* tradicional, não se sabe se qualquer uma das linguagens criadas mais recentemente melhora realmente algum aspecto da produtividade do programador. O foco deste trabalho é no subconjunto destas linguagens que são orientadas para arquiteturas *multi-core*.

O objetivo deste projeto é proporcionar uma comparação detalhada das diferentes abordagens, discutir seus pontos fortes e fracos e identificar as áreas de aplicação às quais são mais adequadas. Isso exigiu a resolução de problemas relevantes (do ponto de vista de concorrência) utilizando cada abordagem. Também foi necessário decidir sobre a metodologia da comparação.

Os resultados incluem: um levantamento de trabalhos relacionados, uma metodologia para comparar linguagens de programação concorrente, e resultados do estudo mostrando a aplicabilidade da metodologia, onde 6 linguagens são comparadas em termos de tempo para codificar, tamanho do código fonte, tempo de execução e *speedup*, através da solução de 6 problemas.

Palavras-chave: Concorrência, usabilidade, linguagens de programação.

1 INTRODUCTION

The recent transition of general purpose computing to multi-core architectures has drastically changed the computing field. This transition was caused by the physical constraints preventing frequency scaling (ASANOVIC et al., 2009). It is not the case anymore that the next generation of hardware will make the programs much faster, unless they are created with concurrency in mind – a task that has long been recognized as difficult and error-prone.

The main reasons why this task is so difficult are: race conditions, dead/live locks, starvation and parallel slow down (explained in more depth in section 2.1.2). Those bugs are not familiar to the sequential programmer. Even worse, they tend to be nondeterministic, which means they are hard to reproduce and thus hard to understand and fix.

Although many programming languages have been proposed to make writing correct concurrent programs simpler than traditional multithreading, it is largely unknown whether any of the newly created languages actually improves some aspect of programmer productivity. This work focuses on the subset of those languages that are oriented towards multi-core architectures.

Those languages try to make the programmer's life easier by introducing high-level abstraction and hiding most of the low level details used to implement them. The problem with those abstractions is that they might remove flexibility that would be necessary to efficiently solve some problems.

1.1 Objectives

The goal of this project is to provide a detailed comparison of the different approaches, discuss their strengths and weaknesses and identify the application areas they are best suited for. The approaches are compared in terms of metrics such as source code size, time to code a solution and speedup. This required solving relevant problems (from a concurrency point of view) using each approach and developing a suitable methodology of comparison.

1.2 Results

The first result is a survey of 125 concurrent programming languages together with their main characteristics (concurrent programming paradigm, communication paradigm, sequential programming paradigm, etc.). The languages compared in this work were chosen from this larger set. A survey on concurrent programming problem sets was also done, in order to choose the problems that would be solved.

The next result is a methodology for comparing concurrent programming languages, including the problems to be solved, metrics to be considered, and a general study design. This methodology was designed based on the comparisons of concurrent programming languages found on the literature.

The final result is the application of the methodology to the comparison of 6 concurrent programming languages (Chapel, Cilk, Erlang, Go, SCOOP, TBB), in terms of time to code, source code size and speed, through the solution of 6 problem from the Cowichan problem set (WILSON; IRVIN, 1995). It was concluded that TBB and Cilk are the best languages for concurrent programming on multi-core architectures, followed closely by Go and Chapel (which still need to have the implementation improved), and then by Erlang and SCOOP (which are more suited to distributed computing).

1.3 Structure of this Document

Chapter 2 is about the relevant general concepts and the literature search (survey of related work). Chapter 3 describes the programming models and problems considered in this work, as well as the chosen methodology. In chapter 4 the implementation of the problems in the various programming models is explained. Chapter 5 presents the results and, finally, chapter 6 presents the conclusions.

2 CONCEPTS

This chapter describes the concepts relevant to this work. The concepts were extracted mainly from Tanenbaum (2007), Andrews; Schneider (1983), Andrews (2000) and Tanenbaum; Steen (2006).

In this work the terms *concurrent programming model* and *concurrent programming paradigm* are used interchangeably to refer to concurrent programming languages and concurrent programming libraries.

2.1 Concurrency

Concurrent computing means that programs are composed of interacting parts that may execute in parallel. It is expected that the programs actually run in parallel when there are multiple processing units, as is the case for multi-core computers. Thus, the terms *concurrent* and *parallel* are used interchangeably in this work.

The communication between those interacting parts may be implicit or explicit. Implicit communication occurs in the form of *futures*, or *promisses*, which are handles to results of concurrent computations that might be yet incomplete. Explicit communication is further divided in:

Shared memory: the components communicate by changing shared memory locations. This usually implies the use of locking for coordination;

Message passing: the communication occurs through the exchange of messages. This exchange can be synchronous or asynchronous. In a synchronous exchange, the sender and the receiver block until they are both ready to communicate. Asynchronous exchange avoid the blocking step by keeping the messages temporarily in buffers.

2.1.1 Why concurrency?

The main reasons for concurrency in general are:

To add or improve fault tolerance: e.g., using other cores for redundant computation;

To increase storage: e.g., more cores usually means more cache memory;

To increase performance: e.g., to increase throughput, to decrease latency, to take advantage of data locality (inside the cache memories).

2.1.2 What can go wrong?

The main caveats of concurrent programming are:

Race conditions: problem where the output of a parallel program changes depending on the exact timing of the execution. It happens when shared-state is modified by more than one thread of execution. It is usually avoided by the use of locks;

Dead/Live locks: problem where a group of threads of execution indefinitely stops progressing. It usually happens when there is a cyclic dependence between the threads regarding the locks they own and the locks they are waiting for;

Starvation: problem where one specific thread of execution indefinitely stops progressing. It usually happens when it competes for a resource with a higher priority thread of execution that always wins;

Parallel slow down: problem where a parallel program is slower than a sequential program to solve the same task. It can happen due to lock contention (when the threads of execution expend too much time waiting for locks) and due to overheads in scheduling and communication.

2.1.3 Concurrent Programming Paradigms

The following is a list of the relevant concurrent programming paradigms:

Partitioned Global Address Space (PGAS): global address space with part of the memory local to each processor (YELICK et al., 2007);

Algorithmic Multithreading: algorithmic expression of concurrency, making programs processor-oblivious (optimal for any number of processors) (BLUMOFE et al., 1995);

Actor Model: actors send asynchronous messages to other actors (AGHA, 1986);

Communicating Sequential Processes (CSP): parallel composition of a fixed number of sequential processes that communicate synchronously using messages in a fixed pattern (HOARE, 1978);

Concurrent Object-Oriented: object-oriented language extended with concurrency (MORANDI; BAUER, 2010);

Algorithmic Skeletons: use of parallel programming patterns (COLE, 1991).

2.1.4 Concurrent Programming Design Patterns

The main concurrent programming design patterns are:

Parallel For: concurrent variant of for statement, iterates over a range of elements in parallel, evaluating the loop body exactly once for each element, and then waiting for all iterations to finish before starting the next instruction;

Parallel Reduce: concurrent variant of a reduce statement, computes the aggregate value over a range, given a binary aggregation operation. Examples include finding the maximum or the sum of an array or a matrix;

Parallel Scan: concurrent variant of a scan variant, computes all the intermediary values in the aggregation over a range from left to right, given a binary aggregation operation. Examples include finding the prefix maximum or sum (i.e. index i of the answer will contain the maximum or sum in the range from 0 to i).

2.2 Empirical Studies

This section reviews the design of empirical studies, since part of this work is empirical. As said by Perry; Porter; Votta (2000), the main steps of empirical studies are:

1. Hypothesis formulation;
2. Observation;
3. Abstraction of the observation into data;
4. Data analysis;
5. Conclusion elaboration, with respect to the hypothesis.

The authors also mention that studies should strive to establish principles that are casual, actionable and general. They cite the following as components of a good empirical study:

- Research context;
- Hypothesis;
- Experimental design;
- Threats to validity;
- Data analysis and presentation;
- Results and conclusion.

2.3 Comparing Programming Models

This section reviews comparisons of usability, productivity and performance of concurrent programming models. They helped choosing the methodology of this work.

Nanz et al. (2011) present an empirical study to compare the ease of use (program understanding, debugging and writing) of two concurrency programming approaches (SCOOP and Multi-threaded Java). They use self-study to avoid teaching bias, self-evaluation to verify the equivalence of the tested groups, and standard evaluation techniques to avoid subjectivity in the evaluation of the answers. They conclude that SCOOP is indeed easier to use than Multi-threaded Java regarding program understanding and debugging, and equivalent regarding program writing.

Szafron; Schaeffer (1994) experimentally assess the usability of two parallel programming systems (NMP and Enterprise) using a population of 15 students, and one problem (transitive closure). Half of the students used each language. They analyzed 6 statistics: number of hours, lines of code, number of sessions, number of compiles, number of runs, and execution times. The paper was criticized by Selknaes (2009).

Selknaes (2009) discusses the work by Szafron; Schaeffer (1994) and concludes that the population was small and biased (15 students with a shared interest in parallel programming and a shared educational background), that the statistical significance of results should have been mentioned and that the systems might not be comparable since they are very different in nature.

M. Sub (2005) describe the plans (not fulfilled) to evaluate the present state of affairs concerning parallel programming and its systems. The authors propose a survey among programmers and scientists, a comparison of parallel programming systems using a standard set of test programs, and a wiki resource for the parallel programming community.

Bal (1991) is a practical comparative study based on actual programming experience with 5 languages (SR, Emerald, Parlog, Linda and Orca) and 2 problems (traveling salesman problem, all pairs shortest paths). It reports the authors experience while implementing the solutions.

E. Kemal S. Vivek (2011) measure the productivity of 3 parallel programming languages (C + MPI, UPC and X10), using 27 students, and 1 problem (Smith-Waterman local sequence matching). Productivity insights are: abstraction mismatch, lack of performance transparency, lack of programming style and discipline, lack of knowledge of parallel design idioms, nondeterminism considered harmful.

Cavé; Budimlić; Sarkar (2010) is a comparison of library and language-based approaches to parallel programming, supported by the experiences in teaching both models at Rice University. The authors conclude that library-based approach can be verbose and requires a lot of meticulous micro management whereas language-based approach provides enough flexibility to hide complexity as well as reduce the possibilities of flawed designs and errors in programs, therefore a language approach is more suitable for mainstream users who lack expertise in parallelism.

Hochstein et al. (2008) compare programming effort for two parallel programming models (message-passing and PRAM-like), using one problem (sparse-matrix dense-vector multiplication), with two groups of students. The used metrics are: development time and program correctness. The results show a 46% reduction in the mean PRAM-like development time (95% confidence interval) compared to message-passing, and no statistically significant difference in correctness rates.

Hochstein et al. (2005) is a case study of the parallel programmer productivity of novice parallel programmers using HPC classroom environments. The authors consider 2 problems (game of life and grid of resistors), and three programming models (serial, MPI and OpenMP). They investigate speedup, code expansion factor, effort required, and cost per line of code. The work concludes that there is a statistically significant difference between the code expansion rates for MPI and OpenMP.

2.4 Existing Surveys of Programming Models for Concurrency

This section reviews surveys of programming models for concurrency. They helped choosing the languages considered in this work.

Philippsen (2000) is an extensive catalog of 111 Concurrent Object-Oriented Languages and their main features regarding the mechanisms used to initiate and to coordinate concurrency. The advantages, disadvantages and interdependencies of these features are analyzed. The degree of computation and data locality that can be achieved is discussed. Performance is evaluated in terms of known key performance factors (fan-out, intra-object concurrency and locality), not in terms of benchmarks of any kind. The disadvantages of the features are stated in terms of broken encapsulation, inheritance anomalies, expressiveness of coordination constraints, and choice of implementation (language vs library). Programming environments oriented towards distributed programming are not considered.

Wyatt; Kavi; Hufnagel (1992) present a comparison of 14 concurrent object oriented languages regarding communication, synchronization, process management, inheritance and implementation trade-offs. They conclude that none of them met Meyer's seven requirements for a language to be truly object-oriented, because they either restrict or disallow inheritance. The languages were selected based on the availability of published material and detailed manuals. There is no discussion about the application areas the languages are best suited for.

Papathomas (1995) examine the issues underlying concurrent object-oriented programming and how different approaches for language design address these issues. The authors consider the design space, the criteria for evaluating language design choices and them compares the approaches to the design with respect to the criteria.

Feldman (1990) analyze the concurrency constructs (process creation, activation, synchronization, communication, scheduling, termination; non determinism) in 5 programming languages, as well as in the UNIX operation system.

Bal; Steiner; Tanenbaum (1989) discuss 15 distributed languages, focusing on constructions for parallelism, communication, synchronization and fault tolerance. The authors define distributed systems as multiple processor and distributed memory systems. In addition, 4 classes of distributed applications are defined: parallel, high-performance applications; fault-tolerant applications; applications using functional specialization; inherently distributed applications.

Arjomandi; O'Farrell (1992) analyze concurrency issues (memory model, communication, active objects, thread/objects, inheritance) in object-oriented languages, particularly C++-based languages. Concerned with parallel programming and not with distributed programming.

Stotts (1982) compare concurrent programming languages by orientation: synchronous vs asynchronous, shared memory vs message passing vs dataflow (or actor). The authors use two ease of use criteria: excessive syntax and duplicated capabilities, and common syntax with uncommon semantics. Then 13 procedural concurrent languages classified by: communication method, synchronization method, unprotected shared variables, buffers / user defined length, process creation, process topology, code-valued variables, nondeterminate execution / explicit expression, separate compilation, real-time support, procedure / data abstraction, exception handling, and proof support.

Andrews; Schneider (1983) identify major concepts of concurrent programming and describe some of the more important language notations for writing concurrent programs. According to them, the ways to specify concurrent execution are: coroutines, fork / join, cobegin / coend, and process declarations. Synchronization primitives (shared memory) are: exclusion and condition synchronization, implemented using busy-waiting, semaphores, conditional critical regions, monitors, or path expressions. Message-passing primitives are: channel specification, remote procedure calls, and atomic transactions. General classes of concurrent programming languages are: procedure oriented (shared memory), message oriented (send and receive), procedure oriented (remote procedure call).

Al Zain et al. (2008) investigate whether a programming language (Glasgow parallel Haskell) with high-level parallel coordination and distributed shared memory model can deliver good and scalable performance on computational Grids. The authors define a parallel program as being equal to computation plus coordination.

Scaife; As (1996) discuss issues in the design and selection of concurrent object-oriented languages and outlines some general principles, surveying 29 programming languages.

Mitchell; Wellings (1996) review Bloom's criteria for evaluating the expressive power of synchronization primitives, and apply it in an object-oriented framework. The criteria are: type of request, order of request, request parameters, local state, and history information.

Philippsen (1995) is a survey of 98 concurrent object-oriented languages classified by: object-orientation characteristics, memory model, parallelism, scheduling, mapping, synchronization, fault tolerance, and availability.

Briot et al. (1998) discuss three approaches to concurrency and distribution in object-oriented programming, namely library (using class libraries; oriented towards system builders), integrative (merging concepts such as object and activity; oriented towards application builders) and reflective ("bridge" between the two other approaches; and oriented towards both application builders and system builders).

2.5 Relevant Concurrency Problems

This section reviews relevant concurrency problems found on the literature. They served to determine the problems used in this study.

Asanovic et al. (2006) use 13 "Dwarfs" (algorithmic methods that capture a pattern of computation and communication), instead of traditional benchmarks, to design and evaluate parallel programming models and architectures: dense linear algebra, sparse linear algebra, spectral methods, n-body methods, structured grids, unstructured grids, monte carlo, combinatorial logic, graph traversal, graphical models, finite state machines, dynamic programming, and backtrack / branch-and-bound.

Asanovic et al. (2008) present 5 applications as being of high importance in the landscape of parallel computing: personal health, image retrieval, hearing / music, speech, parallel browser.

Asanovic et al. (2009) propose 5 applications: music / hearing, speech understanding, content-based image retrieval, intraoperative risk assessment for stroke patients, parallel browser. The authors argue in favor of: architecting parallel software with design patterns, not just parallel programming languages; splitting productivity and efficiency layers, not just a single general-purpose layer; generating code with search-based autotuners, not compilers; analyzing systems with sketching; verification and testing, not one or the other; parallelism for energy efficiency; energy amortization; energy savings; and space-time partitioning for deconstructed operating systems.

Breitinger; Loogen (1995) give an overview of the main stream of research on concurrency in declarative languages and motivate the definition of the concurrent functional language EDEN.

Wilson (1993) propose 6 problems in order to access the usability of parallel programming systems: Turing ring, kece, active chart parsing, image thinning and skeletonization, polygon overlay, skyline matrix solver.

In Wilson; Bal (1996), the authors determine usability of a parallel programming system through medium size (about 1000 lines of code), realistic applications, the so-called Cowichan Problems. Each problem was implemented by a MSc. student as the MSc. thesis. Wilson; Irvin (1995) say that implementing each Cowichan problem takes six to eight weeks. They propose a suite of toy problems: elastic net, Gaussian elimination, halving shuffle, invasion percolation, game of life, Mandelbrot set, point normalization, outer product, matrix-vector product, random matrix generation, successive over-relaxation, image thresholding, vector difference, point value winnowing. They also call those problems Cowichan problems.

Anvik et al. (2002) assert the utility of CO2P3S programming model using the following problems: Fifteen Puzzle (IDA* search), Kece (Alpha-Beta search), Skyline Matrix Solver (LU-Decomposition), Matrix Product Chain (Dynamic Programming), Map Overlay (Polygon Intersection), Graphics (Image Thinning), Reaction / Diffusion (Gauss-Seidel / Jacobi).

Feo (1992) propose the Salishan problems (Hamming number generation, isomer enumeration, skyline matrix reduction, discrete event simulation) to compare parallel programming languages.

Trono (1994) present Santa Problem (a problem similar to the classical Dining Philosophers problem), created to be an exercise in a systems class solved using semaphores.

Cantonnet et al. (2004) analyze the productivity of 2 languages (UPC and MPI), using 7 problems (conjugate gradient, fast Fourier transform, integer sorting, multigrid, gups, histogram, n-queens).

2.6 Methodology

This section reviews additional methodologies used to compare parallel programming models. They also influenced the methodology used in this work.

Anvik et al. (2002) assert the utility of CO2P3S programming model using the following metrics: lines of code (Sequential, Parallel, Generated, Reused, New), and speedup (2, 4, 8, 16 cores with a fixed problem size).

Wilson; Irvin (1995) propose a suite of toy problems and suggest to first implement each problem as a stand-alone program, then chain the problems together to create the kind of phase behavior seen in real applications and test the support for code re-use, information hiding and heterogeneous parallelism.

Bouman (1995) report the development, experiences and results of the implementation of a skyline matrix solver using Orca parallel programming language.

Jeeva Paudel (2011) investigate the programmability of X10 programing system using the Cowichan problems. They conclude analyzing the strengths and weaknesses of the language. Strengths: flexible treatment of concurrency through lightweight activities, distribution through distributable arrays, and locality through a PGAS memory model, within an integrated type system; more flexible memory model through asynchronous partitioned global address space, and data distribution through various kinds of array constructs. Weaknesses: runtime performs poorly in its handling of fine-grained asynchronous activities; lack of error reporting, debugging and testing framework.

Cantonnet et al. (2004) analyze the productivity of 2 languages (UPC and MPI), using the metrics of lines of code (LoC) and number of characters (NoC).

Teijeiro et al. (2009) present a methodology to accomplish a programmability study, through the use of classroom studies with a group of novice UPC programmers.

Wilson et al. (1992) assess the usability of two parallel programming environments. The assessment factors are: performance (speed of code, memory usage), usability (suitability for large-scale software engineering), availability (portability, hardware dependence). The authors also present a taxonomy for parallel programming systems.

Chamberlain; Callahan; Zima (2007) is an assessment of the parallel programmability of the Chapel language. The authors define principles for productive parallel language design (identification of parallelism, synchronization, data distribution and locality, global view of computation, support for general parallelism, separation of algorithm and implementation, broad-market language features, data abstractions, performance, execution model transparency, portability, interoperability with existing codes, and bells and whistles). They also present a parallel language survey (communication libraries and PGAS languages).

Skillicorn; Talia (1998) is an assessment of the suitability for realistic portable parallel programming of parallel programming models and languages, using six criteria (ease of programming, presence of a software development methodology, architecture-independence, ease of understanding, guarantee of performance, and estimation of cost). It includes a classification of models of parallel computation.

3 PROBLEM

The first step done was to choose the relevant programming models to be analyzed and a relevant problem set to be used in the analysis. The following sections describe and explain those choices.

3.1 Programming Models

The programming models where selected (from a set of 125 that were considered, see Appendix A) based on the following criteria:

- Positive:

  Usage: used for real applications;

  Target architecture: focus on multi-core;

  Variety: adds to the variety of programming paradigms, communication paradigms, and / or concurrency paradigms considered.

- Negative:

  Old: not used anymore;

  Liveness: project discontinued;

  Architecture: not for multi-core;

  Research: no published paper;

  Young: not yet implemented.

The chosen languages were: Chapel, Cilk, Erlang, Go, SCOOP, and TBB. Table 3.1 summarizes their characteristics, and the following subsections describe each of them. Although OpenMP and MPI are the current state of the art in parallel programming tools for shared-memory and distributed-memory, respectively (ANVIK et al., 2002), this work focuses on discovering the languages that have potential to be the next big successes.

3.1.1 Chapel

Chapel is a global-view parallel language that supports a block-imperative programming style (CHAMBERLAIN; CALLAHAN; ZIMA, 2007). Parallelism is described

Table 3.1: Main language characteristics

Name	Concurrency Paradigm	Communication Paradigm	Programming Paradigm	Year	Corporation
Chapel	Partitioned Global Address Space	Message Passing / Shared Memory	Object Oriented	2005	Cray
Cilk	Structured Fork-Join / Algorithmic Multithreading	Shared memory	Structured	1994	Intel
Erlang	Actor	Message Passing	Functional	1986	Ericsson
Go	Communicating Sequential Processes	Message Passing / Shared Memory	Structured	2009	Google
SCOOP	Simple Concurrent Object Oriented Programming	Message Passing	Object Oriented	1993	Eiffel Software
TBB	Skeleton	Shared Memory	Template Library	2006	Intel

in terms of independent computation implemented using threads, but specified through higher-level abstractions. It was designed to improve parallel programming productivity, from multi-core to cluster, and won HPC Most Elegant Language award 2011 (SPEAKER-HUANG, 2011).

The main abstractions used in this work are:

forall: concurrent variant of *for* statement, it provides parallel iteration over a range of elements (INC., 2011). The loop body is evaluated once for each element in the range, each instance possibly being evaluated concurrently with others. The loop must be serializable. Control continues with the statement following the forall loop only after every iteration has been completely evaluated (all data accesses within the body of the forall loop will be guaranteed to be completed);

reduce: implements parallel reduce, collapsing the aggregate's value down to a summary value (e.g. computing the sum of the values in an array);

scan: implements parallel scan, computing an aggregate of results where each result value stores the result of a reduction applied to all of the elements in the aggregate up to that expression (e.g. computing the prefix-sum (LADNER; FISCHER, 1980) of the values in an array).

3.1.2 Cilk

Cilk is a linguistic and runtime technology for programming dynamic multi-threaded applications on shared-memory multiprocessors (BLUMOFE et al., 1995). Parallelism is exposed through high-level primitives that are implemented by the runtime system, which takes care of scheduling and load balancing. The runtime system guarantees efficient and predictable performance, by using dynamic scheduling through work stealing. The language won HPC Best Combination of Elegance and Performance award (DONGARRA; KEPNER, 2006).

The main abstractions used in this work are:

cilk: identifies a function as a *cilk* procedure, capable of being spawned in parallel (CILK 5.4.6 REFERENCE MANUAL, 2001);

spawn: concurrent variant of function call statement, starts the (possibly) concurrent execution of a function;

sync: synchronization statement, waits for the end of the execution of all the functions spawned in the body of the current function. There is a implicitly *sync* statement in the end of all *cilk* procedures;

Cilk_active_size: read-only variable that specifies how many processes Cilk is running on;

Self: read-only variable that specifies the current processor number. Guaranteed to be between 0 and *Cilk_active_size* - 1.

The serial elision (erasing the Cilk keywords) of a Cilk program is always a legal C implementation of the Cilk semantics.

3.1.3 Erlang

Erlang is a functional programming language used to build massively scalable soft real-time systems with requirements on high availability (ARMSTRONG, 2003). Parallelism is expressed using the Actor model, where processes are created as independent threads of execution and then communicate through messages (without shared memory). The language has strong industrial use on real-world applications (ARMSTRONG, 1996).

The main abstractions used in this work are:

spawn: used to create a new Erlang process, which is a new thread of execution. Returns a PID which uniquely identifies the process for future communication (ARMSTRONG et al., 1996);

receive and '!' : used to communicate between processes using their PIDs. The communication is always asynchronous.

3.1.4 Go

Go is a general-purpose programming language designed with systems programming in mind (GOOGLE, 2012a). Parallelism is expressed using an approach based on CSP model, in which shared values are passed around on channels instead of being actively shared by separate threads of executions. It has the potential to be a better choice for systems programming than C++ (ECKEL, 2012).

The main abstractions used in this work are:

go: starts the execution of a function or method call as an independent concurrent thread of control, or *goroutine*, within the same address space (GOOGLE, 2012b);

Channels: provide a mechanism for two concurrently executing functions to synchronize execution and communicate by passing a value of a specified element type. Channels can be synchronous or asynchronous.

3.1.5 Simple Concurrent Object Oriented Programming (SCOOP)

SCOOP is a simple object-oriented programming model for concurrency (MEYER, 1993). Parallelism is expressed using the type system and contracts, having simplicity as the main objective. It currently builds on the Eiffel language (MEYER, 1992).

The main abstractions used in this work are:

separate: type annotation that indicates that calls to methods in an object are executed asynchronously (in a different thread of execution) (MORANDI; BAUER, 2010). Race conditions are prevented by requiring that those types of objects be locked whenever they are manipulated;

require: introduces precondition clauses that, when the target of a clause is a *separate* object, effectively function as wait conditions, blocking the execution until the condition is satisfied.

3.1.6 Threading Building Blocks (TBB)

TBB is a high-level, task-based parallel programming template library for the C++ language (INTEL, 2012). Parallelism is expressed using the Skeleton model, and the runtime system takes care of scheduling and load balancing, using work-stealing (PHEATT, 2008). It is directed towards multi-core desktops.

The main abstractions used in this work are:

parallel_for: performs (possibly) parallel iteration over a range of values (KIM; VOSS, 2011); the iteration is executed in non-deterministic order;

parallel_reduce: computes parallel reduction of an associative operation over a range;

parallel_scan: computes parallel prefix (parallel scan) over a range.

3.2 Programming Problems

The following parallel programming problem sets were considered and discarded:

Cowichan Problems (first version): defined in Wilson (1993), each one was implemented as MSc thesis (WILSON; BAL, 1996) and takes six to eight weeks to implement (WILSON; IRVIN, 1995);

Salishan Problems: defined in Feo (1992), they are inspired in the previous set, and thus imply about the same implementation time;

Dwarf Problems: defined in Asanovic et al. (2006), they are just a categorization, without specific problems;

Berkeley Problems: defined in Asanovic et al. (2009), they are also too complex for the purpose of this work;

Benchmarks: the use of benchmarks, like Linpack (DONGARRA, 1988), was discarded since they are intended to compare only low-level performance, and not high-level usability.

The chosen set of problems was the second version of the Cowichan Problem set (WILSON; IRVIN, 1995). It is a suite of toy problems, meaning that implementing each one takes about one day. The problems comprehend a wide range of parallel programming paradigms, and reasonably simple and efficient implementations should be straightforward to produce. In order to be more representative of real-world applications, besides implementing each problem as a stand-alone program, there is a last problem which is a chaining of all the others, so as to test heterogeneous parallelism, code re-use, information hiding and the concurrent execution of different problems.

The following problem descriptions (taken verbatim from the original paper, Wilson; Irvin (1995)) are included here for the sake of completeness.

3.2.1 Randmat: Random Number Generation

This module fills a matrix with pseudo-random integers. The output is required to be independent of the number of processors used. The inputs to this module are:

nrows, ncols: the number of rows and cols in the matrix;

seed: the random number generation seed.

Its output is:

matrix: an integer matrix filled with random values.

3.2.2 Thresh: Histogram Thresholding

This module performs histogram thresholding on a matrix. Given an integer matrix I and a target percentage p, it constructs a boolean matrix B such that B_{ij} is set if, and only if, no more than p percent of the values in I are bigger than I_{ij}. This module inputs are:

matrix: the integer matrix to be thresholded;

nrows, ncols: the number of rows and columns in the matrix and mask;

percent: the minimum percentage of cells to retain.

Its output is:

mask: a boolean matrix filled with $true$ (showing a cell that is kept) or $false$ (showing a cell that is discarded).

3.2.3 Winnow: Weighted Point Selection

This module converts a matrix of integer values to a vector of points, represented as x and y coordinates. Its inputs are:

matrix: an integer matrix, whose values are used as masses;

mask: a boolean matrix showing which points are eligible for consideration;

nrows, ncols: the number of rows and columns in the matrix;

nelts: the number of points to select.

Its output is:

points: a vector of (x, y) points.

Each location where $mask$ is $true$ becomes a candidate point, with a weight equal to the integer value in $matrix$ at that location and x and y coordinates equal to its row and column indices. These candidate points are then sorted into increasing order by weight, and $nelts$ evenly-spaced points selected to create the result vector.

3.2.4 Outer: Outer Product

This module turns a vector containing point positions into a dense, symmetric, diagonally dominant matrix by calculating the distances between each pair of points. It also constructs a real vector whose values are the distance of each point from the origin. Inputs are:

points: a vector of (x, y) points, where x and y are the point's position;

nelts: the number of points in the vector, and the size of the matrix along each axis.

Its outputs are:

matrix: a real matrix, whose values are filled with inter-point distances;

vector: a real vector, whose values are filled with origin-to-point distances.

Each matrix element M_{ij} such that $i \neq j$ is given the value d_{ij}, the Euclidean distance between point i and point j. The diagonal values M_{ii} are then set to $nelts$ times the maximum off-diagonal value to ensure that the matrix is diagonally dominant. The value of the vector element v_i is set to the distance of point i from the origin, which is given by $\sqrt{x_i^2 + y_i^2}$.

$$\begin{tikzcd}
{} \arrow[d, "seed"] & {} \arrow[d, "percent"] & {} \arrow[d, "nelts"] & & & \\
randmat \arrow[r, "matrix"] \arrow[rr, bend right, "matrix"'] & thresh \arrow[r, "mask"] & winnow \arrow[r, "points"] & outer \arrow[r, bend left, "matrix"] \arrow[r, bend right, "vector"'] & product \arrow[r, "result"] & {}
\end{tikzcd}$$

Figure 3.1: Chaining of problems

3.2.5 Product: Matrix-Vector Product

Given a matrix A and a vector V, this module calculates the product $A \cdot V$. Inputs are:

matrix: a real matrix A;

actual: a real vector V;

nelts: the number of values in the vector, and the size of the matrix along each axis.

The output of this module is:

result: a real vector, whose values are the result of the product.

3.2.6 Chain: Chaining Solutions

This problem is the chaining of the previous problems, as shown on figure 3.1. Inputs are:

nelts: the number of elements (used as the dimension of all the matrices and vectors);

randmat_seed: the random number generation seed (for problem *randmat*);

thresh_percent: the minimum percentage of cells to retain (for problem *thresh*);

winnow_nelts: the number of points to select (for problem *winnow*).

The output is:

result: a real vector, whose values are the result of the final product (from problem *product*).

3.3 Methodology

The objective is to compare the languages in order to uncover strengths and weaknesses and provide insights about which type of problems they are best at solving. The methodology was inspired in Bal (1991) and Wilson; Bal (1996), using Wilson; Irvin (1995) problem set.

Most of the studies comparing programming languages (e.g. Anvik et al.; Wilson; Bal; Bouman; Jeeva Paudel; Bal (2002; 1996; 1995; 2011; 1991)) are not empirical, and the ones that are (like Szafron; Schaeffer; E. Kemal S. Vivek; Teijeiro et al.; Hochstein et al.; Hochstein et al. (1994; 2011; 2009; 2008; 2005)) use a group of students, which is not the case here. Instead of considering a population of programmers, this study interprets the problems as the population.

3.3.1 Metrics

The following metrics were used:

Lines of code: the number of source lines of code (LoC). This was one of the first commonly accepted productivity metrics (BOEHM, 1981), and remains in regular use (BOEHM et al., 1995). As said in Dugard (1993), no one has ever demonstrated any better measure of program complexity than a simple count of the number of lines in a program;

Number of words: the number of words in the source code (NoW), measured in addition to LoC, since lines vary on complexity (CANTONNET et al., 2004);

Time to code: time to produce a correct implementation;

Time to execute: time to execute the produced implementation;

Speedup: speedup with a fixed problem size.

3.4 Study Design

The main characteristics of the study design are:

- Git as version control system, used to measure the time to produce solution through the commit times (commits of the form "problem-language-variant keyword", where problem $\in$ {randmat, thresh, winnow, outer, product, chain}, language $\in$ {chapel, cilk, cpp, erlang, go, scoop, tbb}, variant $\in$ {seq, par}, and keyword $\in$ {started, done});

- First write a sequential C++ program to compare behavior, then code a sequential and a parallel version in each language;

- In order to be representative of real-world applications, besides implementing each problem as a stand-alone program, also chain them together in a single application (this is the Chain problem);

- Performance can't be neglected, so spend some effort tuning the parallel programs, in order to obtain reasonable speedup.

3.5 Threats to Validity

This section discusses the main threats to the validity of this study.

Kennedy et al. (2004) say that direct implementation time measurement usually turns out to be invalid because it is difficult to factor out individual ability, and because one might be comparing novice programmers in one language against experienced ones in another. This is not a critical factor in this work, since there is only one programmer and he was not experienced in any of the languages.

Faulk et al. (2004) say that LoC must be applied cautiously, since the number of lines of code necessary to implement a particular functionality varies greatly from one programming language to another, and it is also influenced by the quality of the programmer. Taking into consideration, again, that there is only one programmer in this work, the quality of the programmer won't have a big impact in LoC.

Since there is just one programmer, it is possible to argue that the results will have no statistical significance. But considering the problems (instead of the programmers) as population can provide at least some approximation.

Since the programmer is not an expert in any of the languages, it is possible that an expert could achieve better results. This could be addressed by letting experts in each language check that the quality of the solutions.

Since there is such a variety of languages, it is not clear whether or not they are actually comparable. But as long as it is possible to solve a problem in both languages the metric comparison will be meaningful, and in case it is not possible this goes to the list of language weaknesses.

Problem selection bias, a threat to internal validity (MITCHELL; JOLLEY, 2009), is avoided by using an existing problem set, instead of creating a new one. Repeated testing bias, another threat to internal validity, in the form of acquired knowledge of a problem as its solutions are implemented, was minimized by thinking about the solutions in all languages before starting the first implementation.

The fact of the problem set size being smalls threatens external validity (MITCHELL; JOLLEY, 2009), in the sense that it is unclear whether the results generalize to real world problems. This could be improved by considering a bigger set of problems, which was not possible due to time constraints.

4 IMPLEMENTATION

This chapter describes the main characteristics of the problem implementations. In total 72 solutions were implemented (6 languages x 6 problems x 2 versions – one sequential and one parallel), adding up to more than 10000 lines of code. All the code is Free Software and is available online on BitBucket in the address *https://bitbucket.org/kaue/tc*.

Infrastructure for calculating the code and performance metrics was implemented using Test Driven Development (BECK, 2002) through the Python libraries unittest[1] and mox[2] (over 1500 lines of Python code).

4.1 General Considerations

There is no intention of writing similar solutions to each problem. On the contrary, the idea is to write solutions that seem natural for each language.

Although using dynamically allocated local variables is usually better from a Software Engineering point of view (WULF; SHAW, 1973), the consideration for performance forced the decision to use global, statically allocated ones for the main inputs and outputs. Nevertheless, the language information hiding features were used to prevent cross module intervention (mainly in the *chain* problem).

Since it was not possible (due to time constraints) to optimize the parallel grain size for each language in each problem, it was decided to use a matrix line as grain size, whenever possible. This means that each line of the matrix is processed in parallel.

Solutions are shown in the Chapel language because it is the more concise and clear, since it has primitives for all the required directives.

4.2 Directives

Since not all the language have out of the box support for all required parallel programming directives, they were implemented as necessary. This gave rise to the creation of parallel programming patterns in each language, as discussed next.

[1]http://docs.python.org/library/unittest.html
[2]http://code.google.com/p/pymox/

4.2.1 Parallel For

In Chapel and TBB there are already directives (*forall* and *parallel_for*, respectively) for parallel iteration over a range (Listings 4.1 and 4.2).

```
// parallel for on [begin, end), calling work()
proc parallel_for(begin: int, end: int) {
  forall i in begin..(end - 1) do {
    work(i);
  }
}
```

Listing 4.1: Chapel parallel for

```
// parallel for on [begin, end), calling work()
void parallel_for(int begin, int end) {
  typedef tbb::blocked_range<size_t> range;
  tbb::parallel_for(
    range(begin, end),
    [](range r) {
      for (size_t i = r.begin(); i != r.end(); ++i) {
        work(i);
      }
    }
  );
}
```

Listing 4.2: TBB parallel for

In Cilk and Go, since the creation of threads of execution (*cilk procedures* and *goroutines*, respectively) is very cheap, a recursive, tree-like pattern was used, as shown in Listings 4.3 and 4.4. This approach takes time proportional to $\mathcal{O}(\log n)$ to create n threads of execution.

```
// parallel for on [begin, end), calling work()
cilk void parallel_for(int begin, int end) {
  int middle = begin + (end - begin) / 2;
  if (begin + 1 == end) {
    spawn work(begin);
  } else {
    spawn parallel_for(begin, middle);
    spawn parallel_for(middle, end);
  }
}
```

Listing 4.3: Cilk parallel for

```
// parallel for on [begin, end), calling work()
func parallel_for(begin, end int) {
  done := make(chan bool);
  go parallel_for_impl(begin, end);
  // wait for the workers to finish
  for i := 0; i < end - begin + 1; i++ {
    <-done;
  }
}

func parallel_for_impl(begin, end int, done chan bool) {
  if (begin + 1 == end) {
    work(begin);
    done <- true;
  } else {
    middle := begin + (end - begin) / 2;
    go parallel_for_impl(begin, middle, done);
    parallel_for_impl(middle, end, done);
  }
}
```

Listing 4.4: Go parallel for

In Erlang and SCOOP the creation of threads of execution is expensive, so a linear approach was used instead, in order to avoid the creation of intermediary tree nodes and the intermediary copy of results (Listings 4.5 and 4.6). It should be noted that in SCOOP it is necessary to create different classes in order to introduce concurrency.

```
// parallel for on [Begin, End), calling work()
parallel_for(Begin, End) -> parallel_for_impl(Begin, End - Begin + 1).

parallel_for_impl(Begin, N) ->
  Parent = self(),
  join_results(
    [spawn(fun() -> Parent ! {self(), work(Begin + I)} end) ||
      I <- lists:seq(0, N - 1)].

join_results(Pids) -> [receive {Pid, Result} -> Result end || Pid <- Pids].
```

Listing 4.5: Erlang parallel for

```
class MAIN
feature
  -- parallel for on [Begin, End), calling worker.work()
  parallel_for(Begin, End: INTEGER)
  local
    worker: separate PARFOR_WORKER
    workers: LINKED_LIST[separate PARFOR_WORKER]
  do
    create workers.make
    create parfor_aggregator.make(End - Begin + 1)
    across Begin |..| (End - 1) loop
      create worker.make(Begin, parfor_aggregator)
      workers.exte nd(worker)
    end
    workers.do_all(agent launch_parfor_worker)
    parfor_result(parfor_aggregator)
  end

  launch_parfor_worker(worker: separate PARFOR_WORKER)
  do
    worker.live
  end

  parfor_result(aggregator: separate PARFOR_AGGREGATOR)
```

```
    require
      aggregator.is_all_done
    do
    end

    parfor_aggregator: separate PARFOR_AGGREGATOR
  end

  class PARFOR_WORKER
  create make
  feature
    make (begin_: INTEGER;
          aggregator_: separate PARFOR_AGGREGATOR)
    do
      begin := begin_
      aggregator := aggregator_
    end

    live
    do
      work(begin_)
      put_result(aggregator)
    end

    put_result(an_aggregator: separate PARFOR_AGGREGATOR)
    do
      an_aggregator.put
    end

    Begin: INTEGER
    aggregator: separate PARFOR_AGGREGATOR
  end

  class PARFOR_AGGREGATOR
  create make
  feature
    make (n_: INTEGER)
    local
    do
      n := n_
      count := 0
    end

  feature
    put
    do
      count := count + 1
    end

    is_all_done(): BOOLEAN
    do
      Result := count = n
    end

    n, count: INTEGER
  end
```

Listing 4.6: SCOOP parallel for

4.2.2 Parallel Reduce and Parallel Scan

The two other directives, namely Parallel Reduce and Parallel Scan, where also implemented in the languages where they are missing (Cilk, Erlang, Go and SCOOP). The code was omitted here for brevity.

4.3 Randmat: Random Number Generation

This problem asks to fill a matrix with random values. At first sight there are no dependencies, but in reality there is a hidden dependency on the random generation state. This means a naive approach (e.g.: using the *rand()* function in the C standard library) will either have a race condition (in case the function is not thread-safe) or generate a lot of lock contention (in case it is). This problem was solved by implementing a Linear Congruential Generator (LCG) (KNUTH, 1998) and by using one seed per matrix line, so that it is possible to calculate the numbers for each line in parallel. Another possible approach would be to have one seed per thread of execution, but this would make the result dependent on the number of such threads and thus invalidate the solution. Besides that, the solution is based on a parallel for on the lines of the matrix (Listing 4.7).

```
proc randmat(nrows: int, ncols: int, s: int) {
  const LCG_A: int = 1664525;
  const LCG_C: int = 1013904223;
  const RAND_MAX: int = 100;
  forall i in 1..nrows do {
    var seed = s + i;
    for j in 1..ncols do {
      seed = LCG_A * seed + LCG_C;
      matrix[i, j] = abs(seed) % RAND_MAX;
    }
  }
}
```

Listing 4.7: Chapel Randmat solution

4.4 Thresh: Histogram Thresholding

The solution has the following steps:

1. Find to maximum value in the matrix, using a parallel reduce on the whole matrix;
2. Count the number of occurrences of each value. This is done in two steps:
   (a) Calculate the histogram for each line, using a parallel for on the rows of the matrix;
   (b) Merge the histograms, using a parallel for on the entries;
3. Find the threshold, according to the desired percentage;
4. Generating the mask, according to the threshold, again using a parallel for on the rows of the matrix.

Listing 4.8 shows an implementation. The main caveat in this problem is that all the threads of execution want to access the histogram at the same time. So, in order to avoid both race conditions and lock contention, the solution first calculates a separate histogram for each line (this can be done independently by each thread of execution), and then merges the entries in each histogram (this also can be done independently for each entry).

```
proc thresh(nrows: int, ncols: int, percent: int) {
  // 1. Find to maximum value in the matrix, using a parallel reduce on the whole ↩
      matrix;
  var nmax = max reduce matrix;

  // 2. Count the number of occurrences of each value. This is done in two steps:
  //    (a) Calculate the histogram for each line, using a parallel for on the rows of↩
       the matrix;
  forall i in 1..nrows do {
    for j in 1..ncols do {
      histogram[i, matrix[i, j]] += 1;
    }
  }

  //    (b) Merge the histograms, using a parallel for on the entries;
  forall j in 0..nmax do {
    for i in 2..nrows do {
      histogram[1, j] += histogram[i, j];
    }
  }

  // 3. Find the threshold, according to the desired percentage;
  var count: int = (nrows * ncols * percent) / 100;

  var prefixsum: int = 0;
  var threshold: int = nmax;

  for i in 0..nmax do {
    if (prefixsum > count) then break;
    prefixsum += histogram[1, nmax - i];
    threshold = nmax - i;
  }

  // 4. Generating the mask, according to the threshold, again using a parallel for on↩
       the rows of the matrix.
  forall i in 1..nrows do {
    for j in 1..ncols do {
      mask[i, j] = matrix[i, j] >= threshold;
    }
  }
}
```

Listing 4.8: Chapel Thresh solution

4.5 Winnow: Weighted Point Selection

This problem requires the following steps:

1. For each line in the matrix, find the index of its first selected point on the array of points. This is done in two steps:
   (a) Calculate the number of selected points in each line, using a parallel for on the matrix rows;
   (b) Calculate the cumulative number of selected points in each line, using a parallel scan;
2. Copy the selected points to the array of points, using a parallel for on the rows of the matrix;
3. Sort the array;

4. Copy the chosen points to the output, using a parallel for on the number of chosen points (nelts).

```
proc winnow(nrows: int, ncols: int, nelts: int) {
  // 1. For each line in the matrix, find the index of its first selected point on the↩
       array of points. This is done in two steps:
  //    (a) Calculate the number of selected points in each line, using a parallel for↩
       on the matrix rows;
  forall i in 1..nrows do {
    count_per_line[i + 1] = 0;
    for j in 1..ncols do {
      count_per_line[i + 1] += mask[i, j];
    }
  }

  //    (b) Calculate the cumulative number of selected points in each line, using a ↩
       parallel scan;
  var total = + scan count_per_line;

  // 2. Copy the selected points to the array of points, using a parallel for on the ↩
       rows of the matrix;
  forall i in 1..nrows do {
    var count = total[i];
    for j in 1..ncols do {
      if (mask[i, j]) {
        values[count] = (matrix[i, j], (i, j));
        count += 1;
      }
    }
  }

  // 3. Sort the array;
  var n: int = total[nrows + 1];
  QuickSort(values[0..n]);

  // 4. Copy the chosen points to the output, using a parallel for on the number of ↩
       chosen points (nelts).
  var chunk: int = n / nelts;
  forall i in 1..nelts do {
    var ind: int;
    ind = (i − 1) * chunk + 1;
    (, points[i]) = values[ind];
  }
}
```

Listing 4.9: Chapel Winnow solution

The important point here is to avoid lock contention when creating the array of selected points. This is done by first finding the initial index of the first selected point in each line. The rest of the code straight forward (Listing 4.9).

4.6 Outer: Outer Product

This problem is solved simply with a parallel for on the rows of the matrix, as shown in Listing 4.10.

```
proc outer(nelts: int) {
  forall i in 1..nelts do {
    var nmax: real = -1;
    for j in 1..nelts do {
      if (i != j) {
        matrix[i, j] = distance(points[i], points[j]);
        nmax = max(nmax, matrix[i, j]);
      }
    }
    matrix[i, i] = nmax * nelts;
    vector[i] = distance((0, 0), points[i]);
  }
}
```

Listing 4.10: Chapel Outer solution

4.7 Product: Matrix-Vector Product

The solution for this problem consists of a parallel for on the rows of the matrix (Listing 4.11).

```
proc product(nelts: int) {
  forall i in 1..nelts do {
    var sum: real = 0;
    for j in 1..nelts do {
      sum += matrix[i, j] * vector[j];
    }
    result[i] = sum;
  }
}
```

Listing 4.11: Chapel Product solution

4.8 Chain: Chaining solutions

This problem required using the modularity tools of each language (packages, modules, etc.). Some of the difficulties are cited on section 4.10.

4.9 Correctness Tests

All the 72 solutions were tested for correctness using a variety of input files (at least 5 for each problem) and comparing the respective outputs to the known correct outputs (defined by a sequential C++ implementation of each solution). Those testes were automated by using makefiles (to drive the execution) and the program diff (to compare the outputs with the correct ones).

4.10 Problems Encountered

While developing the solutions, those were the main problems encountered:

Chapel: although there is a parallel scan directive in the language, it is not yet implemented in parallel;

Cilk: it is necessary to declare all the local variables used by a function in the beginning of the function body, and the compile error when declaring a variable somewhere else is just "syntax error", so a better error description should be used;

Erlang: first, all the variable names have to start with an upper case letter, and when they start with a lower case letter the error is just "bad match", so a better error message should be used; second, there is no way to make the *read* function ignore whitespaces;

Go: in order to use packages it is mandatory to put the files in a specify folder structure and set an environment variable, but this should be optional;

SCOOP: the function to read integers only reads the first integer in each line, and skips the rest of the line, it should only skip whitespace; in order to control the lock granularity it is necessary to restructure the whole class hierarchy, there should be at least a read-only type of lock to differentiate from a read-write lock; blocking preconditions (wait conditions) can be wrongly interpreted as assertion preconditions depending on the call context, there should be a clear separation;

TBB: it is not possible to specify an upper bound to the number of execution threads without modifying the source code, there should be a standard flag or environment variable.

5 RESULTS

This chapter presents the results of the work, in terms of the metrics defined in the section 3.3. Extra graphs can be found on appendix D. The performance tests were run in a machine with the following characteristics:

Operating System: Linux 2.6.18-308.1.1.el5 #1 SMP Fri Feb 17 16:51:01 EST 2012 x86_64 GNU/Linux;

Processor: 8x Quad-Core AMD Opteron(tm) Processor 8384 (6M Cache, 2.7 GHz, 100G RAM), total of 32 cores;

Chapel: chpl Version 1.4.0;

Cilk: cilkc Rev: 2491;

Erlang: Erlang R13B03 (erts-5.7.4);

Go: go version go1.0.1;

SCOOP: ISE EiffelStudio version 7.0.8.8074 GPL Edition - linux-x86;

TBB: tbb40_20120408oss, g++-4.5 (GCC) 4.8.0 20120416 (experimental).

Each performance test was repeated 30 times, and the average of the results was taken. All the tests use the same input (nrows = 20000, ncols = 20000, seed = 666, percent = 1, nelts = 10000).

It was verified that the content of matrices doesn't matter when measuring the executing time of any of the problems, except for Winnow, because they execute the same number of operations independently of the matrix content. For Winnow, the proportion of ones in the mask matrix matters, but it only has small impact on the overall performance. Regarding the sizes of the matrices, it was constated that, as expected, the bigger the size, the better the scalability, so the biggest size that fits on RAM memory was chosen.

Another important factor is that for all problems, except Randmat, the input / output time is much bigger than the processing time, since they involve reading / writing matrices to / from disk, respectively. So, in order for the speedup to become apparent, a special flag *is_bench* was added to every solution. This flag means that both input and output should not occur, and the input matrices should be generated on-the-fly instead, which is much faster.

The statistical significance of the results was calculated using the R software environment for statistical computing and graphics[1], version 2.10.1. The p-values can be found on appendix B.

5.1 Time to Code

Figure 5.1 shows the relative time to code each problem in each language with respect to the smallest time to code that problem in any language both for the sequential versions (in 5.1(a)) and the parallel ones (in 5.1(b)). The parallel versions were based on the sequential ones, therefore the parallel time includes the sequential time.

Regarding the sequential versions, the language Cilk took the least time to code in 4 out of the 6 problems. It took almost as little time as Go in the Thresh problem. But it took more than 3 times more to solve the Chain problem, due to name clashes in the functions in each individual solution, which hints on a modularity problem.

Chapel, on the other hand, took the least time to code the Chain problem, mainly because since it already has all the directives implemented (see Sections 4.2 and 3.1.1) the individual problem solutions had less in common. Chapel was also among the smallest time to code on all the other problems, except for Randmat. This is probably due to some difficulty when getting started with the language, since Randmat was the first problem to be solved.

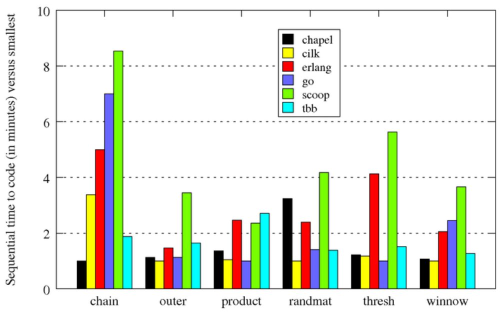
SCOOP took the most time to code in 5 out of the 6 problems, taking from about 3 to more than 8 times more time to code. This was caused in part by the lack of examples and tutorials online, and by its much slower compiling time.

Regarding the parallel versions, Go and TBB took at most 2 times more time to code, and Cilk and Erlang at most 3. Each of those languages took the least time to code in at least one problem, so they seem to be at about the same level of ease of parallel programming, followed closely by Chapel.

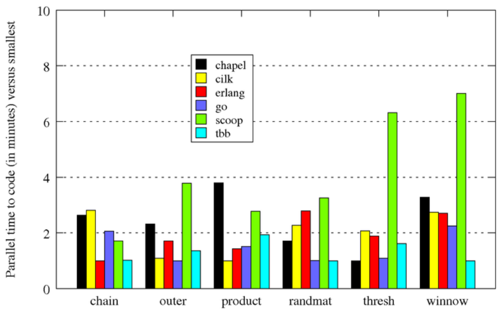
On the other side of the scale, SCOOP took the most time to code in 4 problems, ranging from about 4 to about 7 times more. This is caused by the necessity of creating new classes in order to introduce concurrency (compare the SCOOP code to implement a Parallel For with the respective code for other languages, as shown in section 4.2.1).

The results were compared with 90% confidence using Student t-test with paired observations (JAIN, 1991). Specifically, each language was compared with each other language across all problems in each graph (figures 5.2(a) and 5.2(b)), and across all problems in both graphs (figure 5.2(c)). In those figures, one arrow between two languages means that 0 is not in the 90% confidence interval for the average difference of the language's individual time to code across all problems, and that this interval has only positive values (and thus the source language takes on average more time to code than the destination one across all problems). Although not explicitly shown on the figures (for clarity), this ordering relation is transitive.

[1]http://www.r-project.org/



(a) sequential version



(b) parallel version

Figure 5.1: Ratio of time to code over the smallest

$$\begin{tikzcd}
& scoop \arrow[d] \arrow[ddr] & \\
& erlang \arrow[dl] \arrow[d] & \\
chapel & tbb \arrow[d] & go \arrow[dl] \\
& cilk &
\end{tikzcd}$$

(a) Ordering by sequential time to code (from slower to faster)

$$\begin{tikzcd}
& scoop \arrow[d] \arrow[dr] & \\
chapel \arrow[d] & cilk \arrow[dl] \arrow[d] & erlang \arrow[dl] \\
go & tbb &
\end{tikzcd}$$

(b) Ordering by parallel time to code (from slower to faster)

$$\begin{tikzcd}
& scoop \arrow[dl] \arrow[d] \arrow[dr] & \\
chapel & cilk \arrow[d] & erlang \arrow[dl] \arrow[d] \\
& tbb & go
\end{tikzcd}$$

(c) Ordering by both times to code (from slower to faster)

$$\begin{tikzcd}
scoop \arrow[r] & go \arrow[r] & cilk \arrow[r] & tbb \arrow[r] \arrow[dr] & erlang \\
& & & & chapel
\end{tikzcd}$$

(d) Ordering by source code size (from bigger to smaller)

Figure 5.2: Orderings (with 90% confidence)

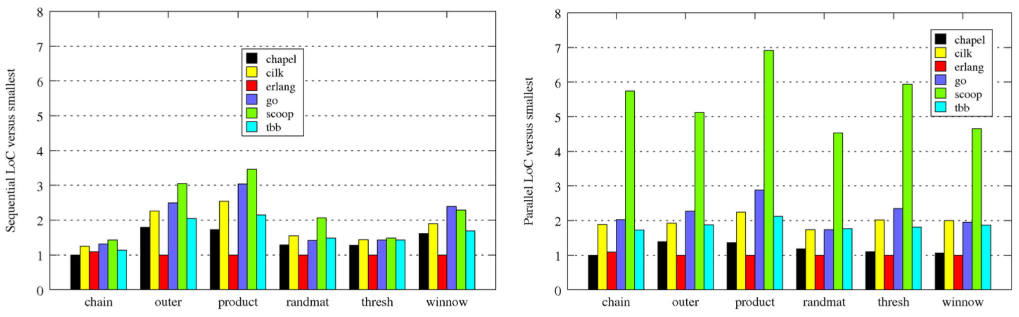
5.2 Source Code Size

The graphs on figures 5.3 and 5.4 show the relative number of lines of source code (LoC) and the number of words (NoW) of each problem solution in each language with relation to the smallest such number in each problem, both for the sequential and the parallel versions. Notice that the numbers in the chain problem are not just the aggregation of all the other problems, because the intermediary input / output functions were removed and, whenever possible, code from one problem was reused on the others.

SCOOP is clearly the more verbose language, both in terms of LoC and NoW. As mentioned in the previous section, this is due to the fact the it imposes the creation of classes to introduce concurrency.

Unsurprisingly, Chapel is among the smallest in most of the problems, mainly because it comes with the parallel directives built-in. What is surprising is that Erlang is as small as, and sometimes smaller than Chapel, even though it doesn't have the parallel directives out of the box. The reason is that being a functional language makes it possible to express the algorithms much more concisely. All the other languages are similar in code size, being at most 2 times bigger on most of the problems.

The results were compared with 90% confidence using Student t-test with paired observations (JAIN, 1991). Specifically, each language was compared with each other language across all problems and all the situations in the four graphs (figure 5.2(d)). In the figure, one arrow between two languages means that 0 is not in the 90% confidence interval for the average difference of the language's individual source code size across all problems, and that this interval has only positive values (and thus the source language has on average a bigger code size than the destination one across all problems). Although not explicitly shown on the figure (for clarity), this ordering relation is transitive.

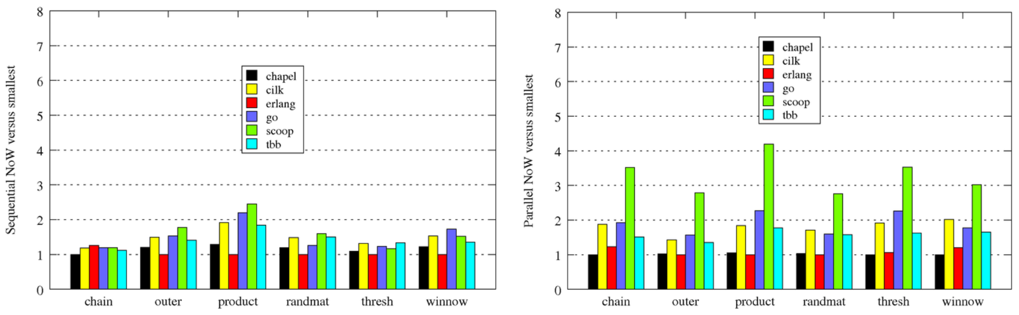


(a) sequential version (b) parallel version

Figure 5.3: Ratio of number of lines of code over the smallest

5.3 Time to Execute

Figure 5.5 shows absolute execution time in seconds for each language and each problem, both for the sequential and parallel (with 8 system threads) versions. The error bars show the 99.9% confidence interval for the mean using Student t-test (JAIN, 1991). Erlang and SCOOP where taken out of the tests because their execution time was more than 10 times bigger than the other 4 languages, making the tests run much slower (see appendix C for more information).



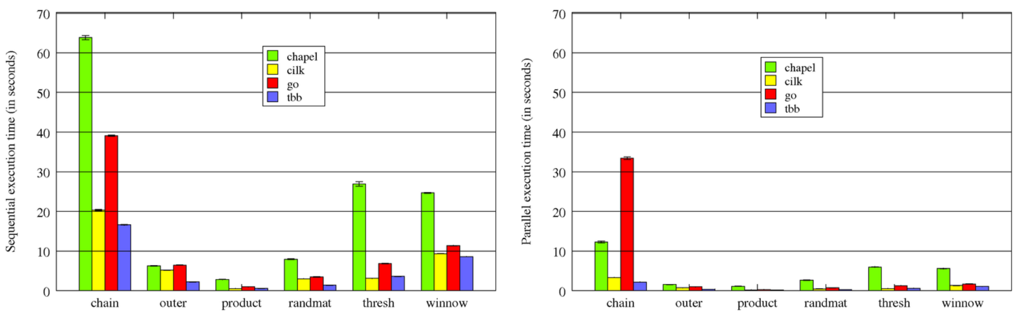
(a) sequential version (b) parallel version

Figure 5.4: Ratio of number of words over the smallest

Chapel took the most time to execute in almost all problems, followed by Go. This seem to be caused by the Chapel compiler being optimized for Grids and Clusters, and not as much for multi-core machines. The bad Go execution time is explained by the language's lack of maturity (only 3 years old), and thus its performance should improve soon. Cilk and TBB where about equally fast on all problems, and faster than Chapel and Go. This is expected, since they are based on the C and C++ languages, respectively.

The results were compared with 99.9% confidence using Welch's t-test with unpaired observations (JAIN, 1991). Specifically, each language was compared with each other language for each problem in each type of graph. In the following paragraph, one language being faster than other in one problem means that 0 is not in the 99.9% confidence interval for the difference of the language's individual mean execution times for that problem, and that this interval has only negative values (and thus the first language takes less time to execute than the second one for that problem).

First, regarding the sequential execution time, it was verified that Go is faster than Chapel in all problems except Outer. Next, regarding the parallel execution time, it was constated that Go is faster than Chapel in all problems except Chain. Finally, regarding both execution time graphs, it was checked that: TBB and Cilk are faster than Go and Chapel in all problems; and TBB is faster than Cilk in all problems except Thresh.



(a) Sequential (b) Parallel (using 8 threads)

Figure 5.5: Execution time (in seconds)

5.4 Speedup

Figures 5.7 and 5.6 show the speedup graphs per problem and per language, respectively, from 1 to 8 (system) threads, with respect to the sequential version of the code (equation 5.1), using the average execution times. Confidence intervals are not shown in order to preserve the clarity of the graphs.

$$S_p = \frac{T_s}{T_p}$$

where: (5.1)

T_s	is the sequential execution time
T_p	is the parallel execution time with p processors
S_p	is the speedup with p processors

None of the languages showed a good speedup for the Product problem. This is due to the fact that the sequential version already takes too little to execute, and the input size can not be increased because it would stop fitting in the RAM memory. So this is a problem with the Product problem, not with any of the languages.

Except for the Product problem, both Cilk and TBB scale very well in all the problems, being at least 6 times faster with 8 system threads. For Go and Chapel the Speedup lines are spread out, showing that they are able to handle some problems better than others.

Go shows a bad performance for the Chain problem, with virtually no speedup. This might be caused by excessive creation of goroutines, generating scheduling and communication overheads. The speedups for the other problems, except for Product, is almost as good as Cilk and TBB, ranging from 5 to 7 when using 8 system threads.

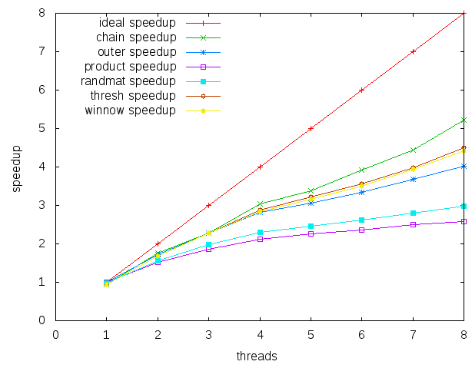
Chapel shows an average performance in every problem, with a speedup going from 3 to 5 in the case of 8 system threads. The language is not specially bad in any of the problems, so it is just missing an overall improvement in its implementation for multi-core architectures.

The results were compared with 99.9% confidence using Welch's t-test with unpaired observations (JAIN, 1991). Specifically, the speedup of each problem for each number of system threads was verified to be statistically significant when compared to the sequential execution time and when compared to the previous speedup (the one with one system thread less).

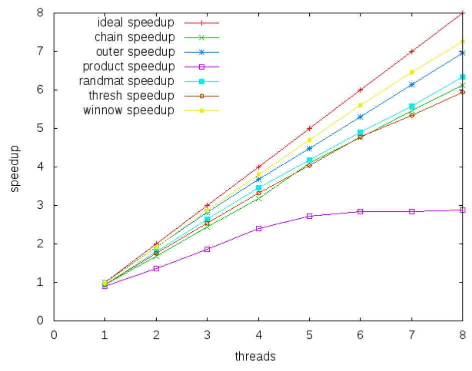
When compared with the sequential execution time, all the speedups were shown to be statistically significant. Regarding the speedup with one thread less, only 4 settings failed the significance test: Go-Chain for 4 and 5 threads (and clearly there is no speedup); Cilk-Product with 6, 7 and 8 threads (which means that the speedup stops growing after 5 threads); and Chapel-Product and TBB-Product with 8 threads (analogous to Cilk-Product).

5.5 Criticism of the Problem Set

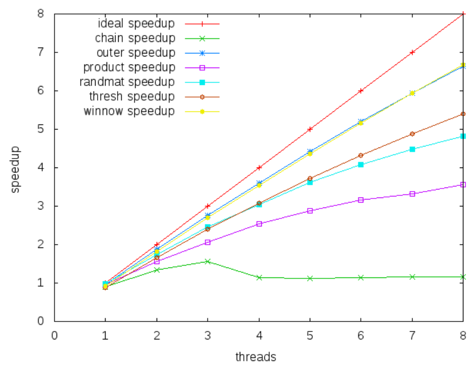
The chosen problems in the Cowichan problem set were discovered to be IO-bound, instead of being CPU-bound, because they require proportionally too little processing in



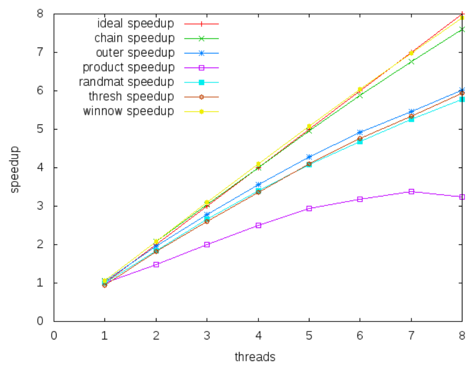
(a) chapel



(b) cilk



(c) go



(d) tbb

Figure 5.6: Speedup per language, in all problems

comparison with the time it takes to read and write the input / output matrices. This made it necessary to change the problem definitions when measuring speedup, in order to decrease the sequential IO part, by generating the matrices on the fly and not outputting the final results.

Another place for improvement in the problem set is that there could be a bigger variety of problems. One particularly good example would be a backtracking / searching problem, which would be far more unbalanced and possibly highlight the advantages of having load balancing as a language feature.

5.6 Discussion

There is no one-size-fits-all solution. The languages have weaknesses and strengths. Table 5.1 shows aggregated values for the three main criteria: time to solution, size of source code and time to execution. The aggregation was done using equation 5.2. In order the emphasize the concurrency features of the languages, the following subset of the metrics was used:

Time to Code: only the time to code the parallel version of each solution was considered, since it is the most relevant and it already includes the time to code the sequential version (as explained on section 5.1).

Size of Source Code: only the size in number of words (NoW) of the parallel version of each solution was considered, since it seems to be more fair than the number of lines of source (LoC) because lines vary on complexity;

Time to Execute: only the time to execute the parallel version with the maximum number of threads (8) was considered, since it is the fastest and includes the effects of the speedup.

$$f(l) = \frac{\displaystyle\sum_{p \in \mathbb{P}} \frac{m(l,p)}{\min\limits_{l' \in \mathbb{L}} m(l',p)}}{|\mathbb{P}|}, \quad l \in \mathbb{L} \tag{5.2}$$

where:

$\mathbb{P}$	is the set of problems
$\mathbb{L}$	is the set of languages
$m : \mathbb{L} \times \mathbb{P} \to (0, \infty)$	is the metric function
$f : \mathbb{L} \to [1, \infty)$	is the aggregated function

The aggregation formula in equation 5.2 calculates for each language the average of the language relative performances in each problem compared to the best performance of any language in the respective problem. Thus, if the language was the best in all problems in a given metric, the result will be 1.0; to give another example, a value of 2.0 for a given language and metric means that, on average, the language was 2 times worse (slower or bigger, depending on the metric) than the best language for that metric in each problem. In the following description, the languages were classified following the criteria on table 5.2.

Table 5.1: Aggregated language metrics

Language	Time to Code	Source Code Size	Time to Execute
Chapel	2.458437	1.011491	7.500411
Cilk	1.996544	1.771280	1.539400
Erlang	1.919658	1.116680	–
Go	1.484067	1.887710	4.985057
SCOOP	4.140901	3.293241	–
TBB	1.320671	1.572715	1.030215

Table 5.2: Criteria for language classification

Range	Classification
$[1.0, 1.5)$	very good
$[1.5, 2.0)$	good
$[2.0, 4.0)$	average
$[4.0, 7.0)$	bad
$[7.0, \infty)$	very bad

Chapel: average time to code, very good source code size, very bad time to execute. Their main focus is on Clusters and Grids, not multi-core computers, but the language has potential for multi-core too;

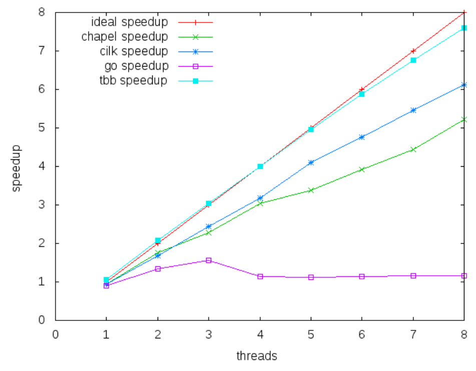
Cilk: good time to code, good source code size, good time to execute. Its abstractions are relatively easy to learn and manipulate, and efficient to execute, probably due to the automatic load balancing using work stealing;

Erlang: good time to code, very good source code size, very bad time to execute. Although it is very concise because of the functional paradigm, it is meant for distributed computing, and it is not yet adequate for multi-core parallel computing;

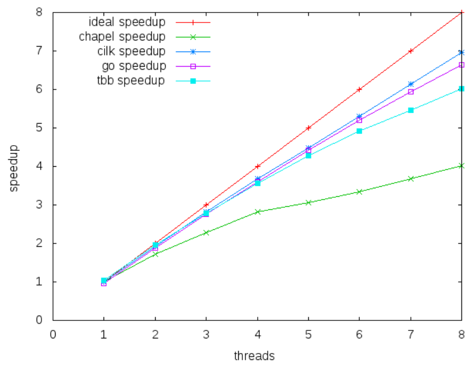
Go: very good time to code, good source code size, bad time to execute. Having channels for communication and synchronization, but at the same time allowing the use of shared memory when necessary make it easy to learn and concise without compromising too much the performance. It is not yet among the best, but as long as the implementation improves it can soon be;

SCOOP: bad time to code, average source code size, very bad time to execute. Although the type system abstracts the concurrency constructs freeing the programmer of the low-level details, it imposes big changes on the design in order to introduce concurrency, and sometimes also removes the freedom to make performance optimizations. Like Erlang, it is probably more suited to distributed computing;

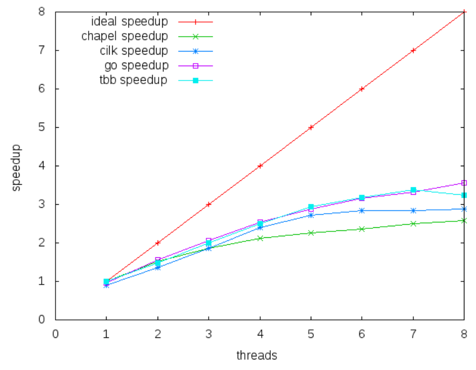
TBB: very good time to code, good source code size, very good time to execute. It is pure a library for C++, which is a well-known language, making it easy to learn; it has all the directives and they can receive lambda function as arguments, making it concise; and it implements load balancing using work stealing, like Cilk, making it very efficient too.



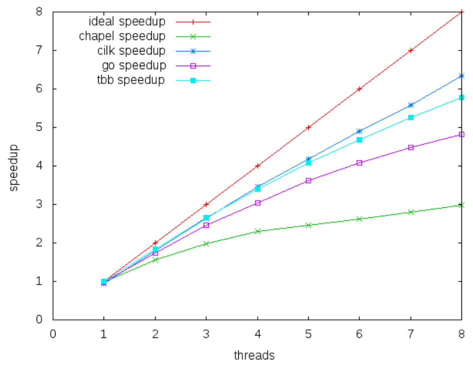
(a) chain



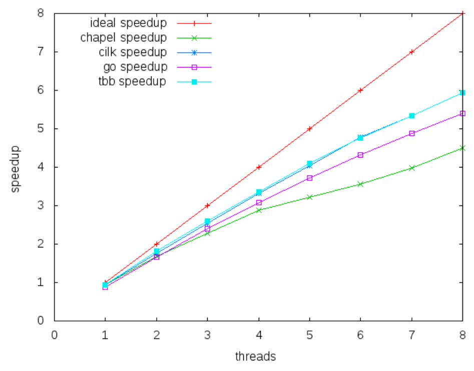
(b) outer



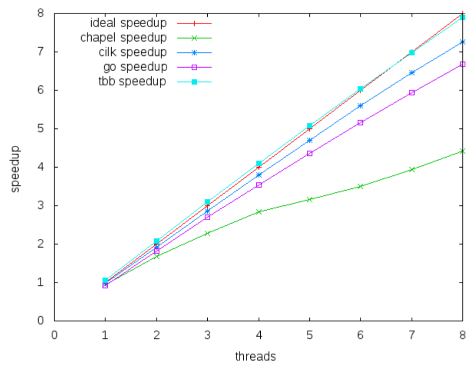
(c) product



(d) randmat



(e) thresh



(f) winnow

Figure 5.7: Speedup per problem, in all languages

6 CONCLUSION

Concurrent programming is necessary and hard, but it is not necessarily hard. It is necessary because physical constraints deter the frequency scaling of single processors, which forces the transition to multi-core ones. It is hard because of all the types of non-deterministic bugs it enables. And yet it is not necessarily hard because of a number of high-level languages and tools that have been proposed to facilitate it.

This work surveyed 125 of those languages, proposed a methodology to compare them, and applied this methodology to a subset of them. The languages Chapel, Cilk, Erlang, Go, SCOOP and TBB where compared in terms of source code size, time to code a solution, time to execute and speedup, through the solution of 6 problems from the Cowichan problem set.

The main conclusion is that the best languages for concurrent programming of multi-core architectures are TBB and Cilk, followed closely by Go and Chapel, and not so closely by Erlang and SCOOP. TBB and Cilk presented small source code size, took little time to code the solutions, and executed fast with a good speedup. Go and Chapel lost in terms of execution time and scalability, showing that there is place for improvements in their implementations. Erlang and SCOOP showed very bad execution time, hinting that they are more suited for distributed programming.

One important lesson learned during the execution of this work is that it is necessary to have performance in mind from the beginning in order to get any speedup. Performance evaluation infrastructure must be in place before the coding of the parallel version is started, so that after parallelizing each loop the gains in performance can be checked. Otherwise there is hardly any speedup, and it is not possible to know which parts need improvement.

Suggestions for future work are: including OpenMP in the set of languages, adding the grain size as parameter to all problems and finding the best grain size for each language, increasing the variety of problems (e.g. by including a backtrack problem), and applying the methodology with a group of programmers.

REFERENCES

AGHA, G. **Actors**: a model of concurrent computation in distributed systems. Cambridge, MA, USA: MIT Press, 1986.

AL ZAIN, A. D. et al. Evaluating a High-Level Parallel Language (GpH) for Computational GRIDs. **IEEE Trans. Parallel Distrib. Syst.**, Piscataway, NJ, USA, v.19, p.219–233, February 2008.

ANDREWS, G. R. **Foundations of Multithreaded, Parallel, and Distributed Programming**. University of Arizona, USA: Wesley, 2000.

ANDREWS, G. R.; SCHNEIDER, F. B. Concepts and Notations for Concurrent Programming. **ACM Comput. Surv.**, New York, NY, USA, v.15, p.3–43, March 1983.

ANVIK, J. et al. **Asserting the utility of CO2P3S using the Cowichan problems**. [S.l.: s.n.], 2002.

ARJOMANDI, E.; O'FARRELL, W. Concurrency issues in C++. In: CENTRE FOR ADVANCED STUDIES ON COLLABORATIVE RESEARCH - VOLUME 1, 1992. **Proceedings...** IBM Press, 1992. p.347–358. (CASCON '92).

ARMSTRONG, J. Erlang - A survey of the language and its industrial applications. In: IN PROCEEDINGS OF THE SYMPOSIUM ON INDUSTRIAL APPLICATIONS OF PROLOG (INAP96). 16 18. **Anais...** [S.l.: s.n.], 1996.

ARMSTRONG, J. **Making reliable distributed systems in the presence of sodware errors (Armstrong)**. 2003. 477+p. Tese (Doutorado em Ciência da Computação) — Swedish Institute of Compuster Science. (3).

ARMSTRONG, J. et al. **Concurrent Programming in Erlang, Second Edition**. 2.ed. [S.l.]: Prentice-Hall, 1996.

ASANOVIC, K. et al. **The Landscape of Parallel Computing Research**: a view from berkeley. [S.l.]: EECS Department, University of California, Berkeley, 2006. (UCB/EECS-2006-183).

ASANOVIC, K. et al. **The Parallel Computing Laboratory at U.C. Berkeley**: a research agenda based on the berkeley view. [S.l.]: EECS Department, University of California, Berkeley, 2008. (UCB/EECS-2008-23).

ASANOVIC, K. et al. A view of the parallel computing landscape. **Commun. ACM**, New York, NY, USA, v.52, p.56–67, Oct. 2009.

BAL, H. E. A Comparative Study Of Five Parallel Programming Languages. In: IN DISTRIBUTED OPEN SYSTEMS. **Anais...** IEEE, 1991. p.209–228.

BAL, H. E.; STEINER, J. G.; TANENBAUM, A. S. Programming languages for distributed computing systems. **ACM Comput. Surv.**, New York, NY, USA, v.21, p.261–322, September 1989.

BECK. **Test Driven Development**: by example. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

BLUMOFE, R. D. et al. Cilk: an efficient multithreaded runtime system. In: JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING. **Anais...** [S.l.: s.n.], 1995. p.207–216.

BOEHM, B. et al. Cost Models for Future Software Life Cycle Processes: cocomo 2.0. In: ANNALS OF SOFTWARE ENGINEERING. **Anais...** [S.l.: s.n.], 1995. p.57–94.

BOEHM, B. W. **Software Engineering Economics**. 1st.ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981.

BOUMAN, D. S. **Parallelizing a Skyline Matrix Solver using Orca**. 1995.

BREITINGER, S.; LOOGEN, R. **Concurrency in Functional and Logic Programming**. 1995.

BRIOT, J.-P. et al. **Concurrency and Distribution in Object-Oriented Programming**. 1998.

CANTONNET, F. et al. Productivity Analysis of the UPC Language. In: IN IPDPS 2004 PMEO WORKSHOP. **Anais...** [S.l.: s.n.], 2004.

CAVé, V.; BUDIMLIć, Z.; SARKAR, V. Comparing the usability of library vs. language approaches to task parallelism. In: EVALUATION AND USABILITY OF PROGRAMMING LANGUAGES AND TOOLS, New York, NY, USA. **Anais...** ACM, 2010. p.9:1–9:6. (PLATEAU '10).

CHAMBERLAIN, B.; CALLAHAN, D.; ZIMA, H. Parallel Programmability and the Chapel Language. **Int. J. High Perform. Comput. Appl.**, Thousand Oaks, CA, USA, v.21, n.3, p.291–312, Aug. 2007.

CILK 5.4.6 Reference Manual. [S.l.]: Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science, 2001.

COLE, M. **Algorithmic skeletons**: structured management of parallel computation. Cambridge, MA, USA: MIT Press, 1991.

DONGARRA, J. The LINPACK Benchmark: an explanation. In: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 1., London, UK, UK. **Proceedings...** Springer-Verlag, 1988. p.456–474.

DONGARRA, J.; KEPNER, J. The 2006 HPC challenge awards. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2006., New York, NY, USA. **Proceedings...** ACM, 2006. (SC '06).

DUGARD, P. Software metrics: a rigorous approach, norman e. fenton. chapman & hall (london) 1991. isbn 0 412 48440 0. price £19.95 (paperback). **Journal of Software Maintenance: Research and Practice**, [S.l.], v.5, n.1, p.59–59, 1993.

E. KEMAL S. VIVEK, E. G. T. **An Experiment in Measuring the Productivity of Three Parallel Programming Languages**. Austin, USA: IEEE Computer Society, 2011. 30–37p.

ECKEL, B. **Calling Go from Python via JSON-RPC**. 2012.

FAULK, S. et al. Measuring HPC productivity. **International Journal of High Performance Computing Applications**, [S.l.], v.2004, p.459–473, 2004.

FELDMAN, M. B. **Language and System Support for Concurrent Programming**. 1990.

FEO, J. T. **Comparative Study of Parallel Programming Languages**: the salishan problems. New York, NY, USA: Elsevier Science Inc., 1992.

GOOGLE. **Effective Go - The Go Programming Language**. 2012.

GOOGLE. **The Go Programming Language Specification - The Go Programming Language**. 2012.

HOARE, C. A. R. Communicating sequential processes. **Commun. ACM**, New York, NY, USA, v.21, n.8, p.666–677, Aug. 1978.

HOCHSTEIN, L. et al. Parallel Programmer Productivity: a case study of novice parallel programmers. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2005., Washington, DC, USA. **Proceedings. . .** IEEE Computer Society, 2005. p.35–. (SC ’05).

HOCHSTEIN, L. et al. A pilot study to compare programming effort for two parallel programming models. **J. Syst. Softw.**, New York, NY, USA, v.81, p.1920–1930, November 2008.

INC., C. **Chapel specification**. Seatle, WA: Cray Inc., 2011.

INTEL. **Deliver Scalable and Portable Parallel Code**: intel threading building blocks. 2012.

JAIN, R. K. **The Art of Computer Systems Performance Analysis**: techniques for experimental design, measurement, simulation, and modeling. 1.ed. [S.l.]: Wiley, 1991.

JEEVA PAUDEL, J. N. A. **Using the Cowichan Problems to Investigate the Programmability of X10 Programming System**. San Jose, CA, USA, 2011.

KENNEDY, K. et al. Defining and measuring the productivity of programming languages. **The International Journal of High Performance Computing Applications, (18)4, Winter**, [S.l.], v.2004, p.441–448, 2004.

KIM, W.; VOSS, M. Multicore Desktop Programming with Intel Threading Building Blocks. **Software, IEEE**, [S.l.], v.28, n.1, p.23 –31, jan.-feb. 2011.

KNUTH, D. E. **The art of computer programming, volume 3**: (2nd ed.) sorting and searching. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

LADNER, R. E.; FISCHER, M. J. Parallel Prefix Computation. **J. ACM**, New York, NY, USA, v.27, n.4, p.831–838, Oct. 1980.

M. SUB, C. L. **Evaluating the state of the art of parallel programming systems**. Universitat Kassel, FB 16, Elektrotechnik/Informatik, 2005.

MEYER, B. **Eiffel**: the language. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992.

MEYER, B. Systematic concurrent object-oriented programming. **Commun. ACM**, New York, NY, USA, v.36, n.9, p.56–80, Sept. 1993.

MITCHELL, M. L.; JOLLEY, J. M. **Research Design Explained**. So Davis Drive Belmont, CA 940002: Wadsworth Cengage Learning, 2009.

MITCHELL, S.; WELLINGS, A. Synchronisation, concurrent object-oriented programming and the inheritance anomaly. **Computer Languages**, [S.l.], v.22, n.1, p.15 – 26, 1996.

MORANDI, B.; BAUER, S. S. **SCOOP – A contract-based concurrent object-oriented**. 2010.

NANZ, S. et al. Design of an Empirical Study for Comparing the Usability of Concurrent Programming Languages. In: INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM'11), 5. **Proceedings...** IEEE Computer Society, 2011.

PAPATHOMAS, M. Concurrency in Object-Oriented Programming Languages. In: NIERSTRASZ, O.; TSICHRITZIS, D. (Ed.). **Object-Oriented Software Composition**. New York, NY, USA: Prentice-Hall, 1995. p.31–68.

PERRY, D. E.; PORTER, A. A.; VOTTA, L. G. Empirical studies of software engineering: a roadmap. In: ICSE - FUTURE OF SE TRACK. **Anais...** [S.l.: s.n.], 2000. p.345–355.

PHEATT, C. Intel threading building blocks. **J. Comput. Sci. Coll.**, USA, v.23, n.4, p.298–298, Apr. 2008.

PHILIPPSEN, M. **Imperative Concurrent Object-Oriented Languages**: an annotated bibliography. [S.l.]: International Computer Science Institute, Berkeley, 1995. (TR-95-049).

PHILIPPSEN, M. A survey of concurrent object-oriented languages. **Concurrency: Practice and Experience**, [S.l.], v.12, n.10, p.917–980, 2000.

SCAIFE, N. R.; AS, E. E. **A Survey of Concurrent Object-Oriented Programming Languages**. 1996.

SELKNAES, J. R. **A survey on usability of parallel programming systems**. 2009.

SKILLICORN, D. B.; TALIA, D. Models and Languages for Parallel Computation. **ACM COMPUTING SURVEYS**, [S.l.], v.30, p.123–169, 1998.

SPEAKER-HUANG, J.-H. SC 2011 Keynote. In: INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS, 2011., New York, NY, USA. **Proceedings...** ACM, 2011. (SC '11).

STOTTS, P. D. A comparative survey of concurrent programming languages. **SIGPLAN Not.**, New York, NY, USA, v.17, p.50–61, October 1982.

SZAFRON, D.; SCHAEFFER, J. Experimentally Assessing the Usability of Parallel Programming Systems. In: IN PROGRAMMING ENVIRONMENTS FOR MASSIVELY PARALLEL DISTRIBUTED SYSTEMS. **Anais...** Birkhauser Verlag, 1994. p.195–201.

TANENBAUM, A. S. **Modern Operating Systems**. 3rd.ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.

TANENBAUM, A. S.; STEEN, M. v. **Distributed Systems**: principles and paradigms (2nd edition). Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

TEIJEIRO, C. et al. Evaluation of UPC programmability using classroom studies. In: THIRD CONFERENCE ON PARTITIONED GLOBAL ADDRESS SPACE PROGRAMING MODELS, New York, NY, USA. **Proceedings...** ACM, 2009. p.10:1–10:7. (PGAS '09).

TRONO, J. A. A new exercise in concurrency. **SIGCSE Bull.**, New York, NY, USA, v.26, p.8–10, September 1994.

WILSON, G. V. Assessing the Usability of Parallel Programming Systems: the cowichan problems. In: IN PROCEEDINGS OF THE IFIP WORKING CONFERENCE ON PROGRAMMING ENVIRONMENTS FOR MASSIVELY PARALLEL DISTRIBUTED SYSTEMS. **Anais...** [S.l.: s.n.], 1993. p.183–193.

WILSON, G. V.; BAL, H. E. **The Cowichan Experience**. 1996.

WILSON, G. V. et al. **Enterprise in Context**: assessing the usability of parallel programming environments. 1992.

WILSON, G. V.; IRVIN, R. B. **Assessing and Comparing the Usability of Parallel Programming Systems**. [S.l.: s.n.], 1995.

WULF, W.; SHAW, M. Global variable considered harmful. **SIGPLAN Not.**, New York, NY, USA, v.8, n.2, p.28–34, Feb. 1973.

WYATT, B. B.; KAVI, K.; HUFNAGEL, S. Parallelism in Object-Oriented Languages: a survey. **IEEE Softw.**, Los Alamitos, CA, USA, v.9, p.56–66, November 1992.

YELICK, K. et al. Productivity and performance using partitioned global address space languages. In: PARALLEL SYMBOLIC COMPUTATION, 2007., New York, NY, USA. **Proceedings...** ACM, 2007. p.24–32. (PASCO '07).

APPENDIX A SURVEY OF LANGUAGES

Table A.1 shows the full list of 125 languages considered as candidates for this study, together with their main characteristics. The blank cells occur whenever it was not possible to determine the classification.

Table A.1: Concurrent programming languages

Name	**Concurrency Paradigm**	**Communication Paradigm**	**Programming Paradigm**	**Based On**	**Year**
ABCL/1 ABCL/R ABCL/R2	Actor	Message Passing	Object-Oriented	Common Lisp	1988
ABCL/c+	Actor	Message Passing	Object-Oriented	C	1988
ACOTES	Stream	Stream	pragmas	OpenMP C/C++	2007
ActorScript	Actor	Message Passing	Functional Imperative Logic	Java C# Objective C SystemVerilog	2010
Ada		Message Passing	Object-Oriented	Pascal	1983
Afnix		Shared Memory	Functional Object-Oriented		1999
Agilent VEE	Dataflow	Pins			1998
Alef		Shared Memory Message Passing	Object-Oriented		1995
Alice		Message Passing	Functional	Standard ML	2000
Ambient Talk	Ambient Oriented	Message Passing	Object-Oriented	E	2006
ASSIST	Skeleton	Stream Distributed Shared Memory			2002

Continued on next page

Table A.1 – continued from previous page

Name	Concurrency Paradigm	Communication Paradigm	Programming Paradigm	Based On	Year
Axum	Actor	Message Passing	Object-Oriented	.NET CLR C	2009
Auto-Pipe	Stream			X	2006
Blitzprog	Actor	Message Passing			2008
BMDFM	Dataflow	Shared Memory		C/C++	2002
Brook	Stream	Shared Memory		C	2003
Calcium	Skeleton			Lithium Muskel	
Chapel	PGAS	Message Passing	Object-Oriented	ZPL High-Performance Fortran	2005
Charm++		Message Passing	Object-Oriented	C++	1993
Cilk		Shared Memory		C/C++	1994
Clojure		Transactional Memory	Functional	Lisp	2007
Concurrent Haskell		Shared Memory Transactional Memory	Functional	Haskell	1996
Concurrent ML		Message Passing	Functional	Standard ML	1999
Concurrent Pascal		Shared Memory	Imperative	Pascal	1975
Co-array Fortran		Message Passing	Object-Oriented	Fortran	1990
CO2P3S	Parallel Design Pattern	Message Passing	Object-Oriented	Java C++	1997
CRIME	Coordination	Publishing of Facts	Logic	Prolog	2007
CUDA	SIMD Stream	Shared Memory	Imperative	C Brook	2006
Curry		Shared Memory	Functional Logic	Haskell	1995
Comega	Join Calculus	Message Passing	Structured Imperative Object-Oriented Event-Driven Functional	C# Polyphonic C#	2003
DUP	Coordination	Message Passing (Pipe TCP Stream)			2008
E	Promisse	Message Passing	Capability-Oriented	Java	2006

Continued on next page

Table A.1 – continued from previous page

Name	Concurrency Paradigm	Communication Paradigm	Programming Paradigm	Based On	Year
Ease	CSP	Shared Data Structures (Distributed Memory)			1993
Eden	Skeleton	Message Passing	Functional	Haskell	1996
Emerald	Distributed Objects	Message Passing	Object-Oriented	Concurrent Pascal	1984
EPAS	Skeleton		Object-Oriented	C++	2005
Erlang	Actor	Message Passing	Functional	Prolog	1986
eSkel	Skeleton	Message Passing	Structured	C MPI	2004
F#	Asynchronous Workflow	Shared Memory	Funcional Imperative Object-Oriented	ML	2002
Fanton	Actor	Message Passing	Object-Oriented Functional (closures)	Java C# Ruby	2005
FAUST	Stream	Stream	Functional Algebraic block-diagrams		2002
Fork	PRAM	Shared Memory	Structured	C	1994
Fortress		Shared Memory		Fortram Scala Haskell	2002
Glenda	Coordination	Message Passing	Structured	Linda	1994
Global Arrays	Global Address Space	Shared Memory over Distributed Memory	Library		1994
Go	CSP	Message Passing	Structured	C	2009
Hadoop	Skeleton (MapReduce)	Message Passing	Structured		2004
HDC	Skeleton	Message Passing	Functional	Haskell	2000
High Performance Fortran	Data Parallel	Shared Memory	Structured	Fortran	1993
HOC-SA	Service	Message Passing			2004
Hydra PPS	Events	Shared Memory	Object-Oriented	Java	2006
Humus	Actor	Message Passing	Functional		1997
Io	Actor	Message Passing	Object-Oriented		2002

Continued on next page

Table A.1 – continued from previous page

Name	Concurrency Paradigm	Communication Paradigm	Programming Paradigm	Based On	Year
iScheme	Actor	Message Passing	Functional	Scheme	2011
Janus	Actor	Message Passing	Object-Oriented Logic	Parlog	1990
JavaSpaces	Coordination	Distributed Shared Memory	Object-Oriented	Java	1999
JHDL	Hardware Description		Object-Oriented	Java	2006
JoCaml	Join Calculus	Message Passing	Functional	OCaml	1997
JoinJava	Join Calculus	Message Passing	Object-Oriented	Java	2002
Joule	Dataflow	Message Passing	Structured		1996
Joyce	CSP	Message Passing	Structured	Concurrent Pascal	1987
Limbo	CSP	Message Passing	Structured	C Pascal	1995
Linda	Coordination	Distributed Shared Memory	Structured		1992
LOTOS	Process Calculus	Message Passing	Imperative		1990
Lucid	Dataflow	Stream			1976
Lustre	Dataflow	Stream			1980
Mallba	Skeleton	Message Passing	Library	C++	2002
MAP3S	Parallel Design Pattern	Message Passing		C MPI CO2P3S	2005
MILLIPEDE	Virtual Parallel Machines	Distributed Shared Memory	Middleware		1997
MPI	Procedural Message Passing	Message Passing	Library	C Fortran	1991
Microsoft Visual Programming	Dataflow	Pins	Graphical		2007
Modula-3		Shared Memory	Structured	Pascal	1980
MultiLisp		Shared Memory	Functional	Scheme	1980
Muskel	Skeleton		Library Object-Oriented	Java	2005
NESL	Data-Parallel		Functional	ML	1993
Newsqueak	CSP	Message Passing	Structured	C	1980
Occam	CSP	Message Passing	Structured		1983
Occam-pi	CSP Pi-calculus	Message Passing	Structured	Occam	2005

Continued on next page

Table A.1 – continued from previous page

Name	Concurrency Paradigm	Communication Paradigm	Programming Paradigm	Based On	Year
OpenCL	Task and Data-Parallel	Shared Memory	Kernel	C	2008
OpenHMPP	Directive-Based	Shared Memory	Codelet	C Fortran	2009
OpenMP	Directive-Based	Shared Memory	API	C C++ Fortran	1997
OpenWire	Dataflow	Pins	Graphical		1997
Orc	Kleene Algebra	Message Passing	Functional		2004
Oz		Message Passing	Logic Functional Object-Oriented	Erlang List Prolog	1991
ParaSail		Message Passing	Object-Oriented		2009
Parc	Block-Oriented Parallel Constructs	Shared Memory	Structured	C	1994
Parlog		Shared Memory	Logic	Prolog	1987
Phoenix	MapReduce	Shared Memory	Functional		2007
Pict	Pi-Calculus	Message Passing		ML	1992
pthreads		Shared Memory	Library	C	1995
Ptolemy II	Actor	Message Passing	Library	Java	1996
P3L	Skelenton	Shared Memory	Structured	C	1999
Quaff	Skeleton CSP	Message Passing	Library	C++	2006
Rebeca	Actor	Message Passing	Object-Oriented		2004
SAC	Array Operations	Shared Memory	Functional	C	1994
SALSA	Actor	Message Passing	Object-Oriented	Java	2001
SBASCO	Skeleton		Component		2004
Scala	Actor	Message Passing	Functional Object-Oriented	Java	2003
SCOOP	Simple Concurrent Object Oriented Programming	Message Passing	Object-Oriented	Eiffel	1993
Sequoia++	Memory Hierarchy	Shared Memory	Object-Oriented	C++	2006
Sh		Shared Memory	Structured	C++	2003
SIGNAL	Dataflow				1982

Continued on next page

Table A.1 – continued from previous page

Name	**Concurrency Paradigm**	**Communication Paradigm**	**Programming Paradigm**	**Based On**	**Year**
SISAL	Dataflow	Stream	Functional	VAL	1983
Skandium	Skeleton	Shared Memory		Calcium	2010
SKElib	Skeleton	Stream	Library	SkIE	2000
SkeTo	Skeleton	Message Passing	Library	C++	2006
SkIE	Skeleton	Stream	Coordination		1999
Skil	Skeleton	Shared Memory	Functional	C	1998
StratifiedJS	Explicit	Shared Memory	Object-Oriented	Javascript	2010
StreamIT	Dataflow	Stream	Kernel		2002
SuperPascal	CSP	Message Passing	Structured	Pascal	1993
System Verilog	Dataflow		Structured		2002
Tersus	Dataflow	Message Passing	Visual		
TBB	Skeleton	Shared Memory	Library	C++	2006
THDL++		Signal	Structured	VHDL C++	
Titanium	Explicit	Distributed Shared Memory	Object-Oriented	Java	
UPC	Explicit	Distributed Shared Memory (Partitioned Global Address Space)	Structured	C	1999
Verilog	Dataflow	Wire			1984
VHDL	Dataflow	Wire			1980
Vvvv	Dataflow		Visual		
WaveScript	Stream	Stream	Functional		2006
XC	CSP	Message Passing	Structured	C	2005
XProc	Dataflow	XML Straem	XML Description		2010
X10	Partitioned Global Adress Space	Distributed Shared Memory	Object-Oriented		2004
ZPL	Implicit Data-Parallel		Array	C	1993

APPENDIX B STATISTICAL RESULTS

The following tables show the p-values for the statistical significance tests explained on chapter 5.

Table B.1: p-values for time to code sequential version

language	chapel	cilk	erlang	go	scoop	tbb
chapel	–	2.591e-01	9.197e-01	6.526e-01	9.910e-01	4.860e-01
cilk	7.409e-01	–	9.788e-01	9.312e-01	9.956e-01	9.198e-01
erlang	8.035e-02	2.117e-02	–	1.570e-01	9.972e-01	4.714e-02
go	3.474e-01	6.879e-02	8.430e-01	–	9.819e-01	2.900e-01
scoop	8.952e-03	4.423e-03	2.780e-03	1.809e-02	–	7.135e-03
tbb	5.140e-01	8.019e-02	9.529e-01	7.100e-01	9.929e-01	–

Table B.2: p-values for time to code parallel version

language	chapel	cilk	erlang	go	scoop	tbb
chapel	–	6.818e-01	5.963e-01	1.312e-02	8.871e-01	1.664e-01
cilk	3.182e-01	–	3.081e-01	5.748e-02	9.144e-01	2.651e-02
erlang	4.037e-01	6.919e-01	–	1.005e-01	9.183e-01	4.894e-02
go	9.869e-01	9.425e-01	8.995e-01	–	9.249e-01	4.765e-01
scoop	1.129e-01	8.564e-02	8.170e-02	7.507e-02	–	5.708e-02
tbb	8.336e-01	9.735e-01	9.511e-01	5.235e-01	9.429e-01	–

Table B.3: p-values for time to code both versions

language	chapel	cilk	erlang	go	scoop	tbb
chapel	–	6.123e-01	8.074e-01	1.097e-01	9.479e-01	1.609e-01
cilk	3.877e-01	–	8.185e-01	1.380e-01	9.722e-01	6.038e-02
erlang	1.926e-01	1.815e-01	–	4.439e-02	9.549e-01	8.391e-03
go	8.903e-01	8.620e-01	9.556e-01	–	9.642e-01	3.772e-01
scoop	5.211e-02	2.780e-02	4.511e-02	3.576e-02	–	2.249e-02
tbb	8.391e-01	9.396e-01	9.916e-01	6.228e-01	9.775e-01	–

Table B.4: p-values for size of source code

language	chapel	cilk	erlang	go	scoop	tbb
chapel	–	9.986e-01	7.150e-01	9.993e-01	9.984e-01	9.997e-01
cilk	1.369e-03	–	1.999e-04	9.690e-01	9.976e-01	1.207e-02
erlang	2.850e-01	9.998e-01	–	9.999e-01	9.990e-01	1.000e+00
go	7.282e-04	3.096e-02	1.219e-04	–	9.967e-01	3.822e-03
scoop	1.634e-03	2.406e-03	9.940e-04	3.288e-03	–	2.688e-03
tbb	3.304e-04	9.879e-01	1.836e-05	9.962e-01	9.973e-01	–

Table B.5: p-values for speedup of language chapel vs sequential time

threads	chain	outer	product	randmat	thresh	winnow
2	2.330e-61	1.399e-54	1.452e-42	2.572e-54	3.331e-36	1.005e-88
3	1.083e-76	2.616e-58	1.566e-48	2.206e-64	7.172e-40	6.555e-90
4	2.260e-83	1.160e-61	4.731e-53	1.418e-67	3.068e-42	2.021e-100
5	1.330e-87	4.531e-57	1.911e-51	3.804e-67	2.016e-42	5.817e-94
6	2.345e-69	4.981e-57	3.241e-51	2.698e-69	7.617e-43	1.111e-91
7	2.934e-84	5.303e-61	2.207e-52	3.272e-67	3.987e-43	5.500e-100
8	1.624e-69	1.120e-59	1.198e-50	4.629e-70	1.018e-43	2.787e-89

Table B.6: p-values for speedup of language chapel vs previous parallel time

threads	chain	outer	product	randmat	thresh	winnow
2	8.672e-76	9.883e-75	1.290e-41	5.947e-56	4.143e-57	9.888e-90
3	8.216e-34	5.505e-52	7.041e-29	3.983e-46	7.394e-66	3.907e-67
4	1.108e-35	7.898e-40	3.619e-17	2.568e-30	4.896e-58	9.713e-54
5	1.326e-13	4.551e-17	1.215e-06	1.139e-13	2.953e-36	3.754e-32
6	8.627e-19	9.251e-21	4.417e-05	1.373e-10	4.644e-36	1.312e-36
7	1.385e-21	3.236e-18	2.586e-05	4.636e-10	3.412e-37	1.903e-34
8	2.853e-23	1.639e-15	5.078e-03	5.752e-10	4.473e-36	7.251e-31

Table B.7: p-values for speedup of language cilk vs sequential time

threads	chain	outer	product	randmat	thresh	winnow
2	8.054e-71	3.977e-65	4.569e-37	1.073e-63	1.138e-62	3.914e-103
3	3.355e-69	5.916e-68	7.079e-49	2.084e-66	1.126e-69	3.469e-96
4	4.950e-61	2.211e-59	3.422e-56	1.156e-75	1.513e-80	5.835e-91
5	6.334e-61	1.008e-65	6.422e-58	5.764e-63	2.762e-75	6.305e-92
6	1.676e-60	1.867e-63	1.629e-54	2.860e-70	7.655e-73	6.533e-100
7	3.058e-61	6.672e-61	1.192e-56	2.755e-76	3.217e-75	1.238e-95
8	2.018e-61	1.125e-62	1.682e-59	1.423e-72	9.200e-76	4.933e-101

Table B.8: p-values for speedup of language cilk vs previous parallel time

threads	chain	outer	product	randmat	thresh	winnow
2	6.837e-90	8.809e-82	1.719e-56	2.866e-96	7.202e-83	3.296e-111
3	5.594e-75	5.289e-76	2.073e-41	8.886e-74	1.204e-74	7.406e-83
4	1.323e-56	1.491e-54	8.534e-28	7.168e-54	4.608e-53	8.076e-79
5	3.587e-77	8.203e-52	5.492e-10	2.026e-37	9.023e-40	4.194e-68
6	8.332e-65	3.496e-44	3.985e-03	1.490e-37	6.612e-36	1.279e-55
7	3.243e-67	4.426e-42	4.581e-01	3.224e-24	7.591e-24	4.115e-46
8	1.616e-58	1.278e-35	2.804e-01	6.108e-21	1.825e-19	2.800e-37

Table B.9: p-values for speedup of language go vs sequential time

threads	chain	outer	product	randmat	thresh	winnow
2	7.196e-64	1.666e-80	1.536e-56	3.746e-48	4.393e-56	4.245e-56
3	8.953e-34	6.182e-79	3.177e-65	3.537e-54	1.467e-57	4.922e-65
4	4.935e-27	7.775e-72	7.225e-63	4.119e-52	4.605e-57	6.731e-63
5	1.050e-26	1.980e-68	9.219e-68	2.275e-51	3.461e-57	1.104e-60
6	8.834e-35	3.856e-67	2.043e-60	2.066e-52	8.012e-56	9.367e-63
7	4.278e-37	7.041e-66	1.222e-71	2.073e-48	1.210e-58	5.946e-62
8	2.032e-41	4.579e-68	2.415e-63	1.506e-51	2.894e-56	1.498e-61

Table B.10: p-values for speedup of language go vs previous parallel time

threads	chain	outer	product	randmat	thresh	winnow
2	1.619e-38	1.255e-74	5.877e-54	7.382e-58	7.726e-93	1.219e-59
3	7.348e-20	2.161e-80	8.639e-43	5.595e-63	1.977e-77	6.058e-91
4	1.000e+00	2.690e-69	1.429e-30	9.763e-45	5.438e-68	7.380e-72
5	1.000e+00	2.825e-59	2.087e-20	2.085e-41	9.879e-63	1.530e-65
6	2.217e-04	3.810e-60	2.668e-14	7.388e-31	7.604e-53	2.090e-66
7	5.735e-05	1.454e-56	2.421e-05	4.483e-23	1.967e-43	2.938e-59
8	4.630e-04	3.655e-46	5.423e-07	2.816e-20	2.412e-35	1.680e-55

Table B.11: p-values for speedup of language tbb vs sequential time

threads	chain	outer	product	randmat	thresh	winnow
2	6.464e-59	2.865e-45	2.407e-30	3.107e-46	9.091e-53	3.418e-95
3	9.141e-58	1.081e-49	2.368e-37	2.466e-48	1.592e-62	5.775e-81
4	2.618e-57	2.972e-50	9.896e-43	1.107e-49	3.746e-60	1.141e-81
5	1.107e-57	6.708e-50	2.437e-40	4.100e-49	5.040e-53	2.952e-70
6	3.411e-58	5.930e-50	7.231e-42	1.071e-51	7.094e-54	8.861e-71
7	1.025e-58	8.639e-50	4.966e-42	1.040e-51	8.197e-57	3.408e-70
8	5.551e-59	1.473e-49	8.843e-43	4.444e-51	2.872e-56	4.571e-70

Table B.12: p-values for speedup of language tbb vs previous parallel time

threads	chain	outer	product	randmat	thresh	winnow
2	2.504e-53	2.509e-49	3.783e-41	3.107e-46	7.507e-52	1.204e-103
3	1.997e-74	2.474e-68	1.790e-42	9.131e-61	4.980e-66	4.241e-74
4	4.318e-71	8.481e-52	1.291e-25	1.018e-49	6.392e-50	1.556e-78
5	2.325e-78	4.697e-44	1.144e-16	9.356e-39	7.324e-38	1.060e-56
6	5.741e-74	2.078e-37	1.100e-07	1.104e-27	3.774e-50	3.593e-73
7	4.512e-63	2.267e-29	8.656e-05	2.798e-21	2.466e-33	5.594e-65
8	3.821e-55	6.575e-28	9.924e-01	2.283e-16	2.858e-26	3.381e-58

Table B.13: p-values for sequential execution time in problem chain

language	chapel	cilk	go	tbb
chapel	–	1.000e+00	1.000e+00	1.000e+00
cilk	7.840e-58	–	6.945e-98	1.000e+00
go	6.335e-49	1.000e+00	–	1.000e+00
tbb	4.073e-56	2.143e-59	6.559e-100	–

Table B.14: p-values for parallel execution time in problem chain

language	chapel	cilk	go	tbb
chapel	–	1.000e+00	9.997e-82	1.000e+00
cilk	1.341e-44	–	1.352e-56	1.000e+00
go	1.000e+00	1.000e+00	–	1.000e+00
tbb	2.175e-44	1.450e-83	6.210e-55	–

Table B.15: p-values for sequential execution time in problem outer

language	chapel	cilk	go	tbb
chapel	–	1.000e+00	6.423e-10	1.000e+00
cilk	8.852e-36	–	4.841e-64	1.000e+00
go	1.000e+00	1.000e+00	–	1.000e+00
tbb	3.751e-55	1.232e-84	1.810e-98	–

Table B.16: p-values for parallel execution time in problem outer

language	chapel	cilk	go	tbb
chapel	–	1.000e+00	1.000e+00	1.000e+00
cilk	5.654e-40	–	3.536e-55	1.000e+00
go	5.713e-34	1.000e+00	–	1.000e+00
tbb	3.550e-42	6.882e-56	6.938e-79	–

Table B.17: p-values for sequential execution time in problem product

language	chapel	cilk	go	tbb
chapel	–	1.000e+00	1.000e+00	1.000e+00
cilk	3.413e-44	–	1.892e-62	1.182e-05
go	8.137e-42	1.000e+00	–	1.000e+00
tbb	2.140e-46	1.000e+00	2.973e-53	–

Table B.18: p-values for parallel execution time in problem product

language	chapel	cilk	go	tbb
chapel	–	1.000e+00	1.000e+00	1.000e+00
cilk	3.349e-45	–	1.823e-27	9.986e-01
go	6.948e-40	1.000e+00	–	1.000e+00
tbb	2.390e-41	1.388e-03	4.565e-41	–

Table B.19: p-values for sequential execution time in problem randmat

language	chapel	cilk	go	tbb
chapel	–	1.000e+00	1.000e+00	1.000e+00
cilk	9.296e-50	–	1.076e-32	1.000e+00
go	2.890e-58	1.000e+00	–	1.000e+00
tbb	5.553e-53	1.136e-81	1.793e-56	–

Table B.20: p-values for parallel execution time in problem randmat

language	chapel	cilk	go	tbb
chapel	–	1.000e+00	1.000e+00	1.000e+00
cilk	1.956e-42	–	1.556e-54	1.000e+00
go	7.416e-41	1.000e+00	–	1.000e+00
tbb	1.431e-42	9.467e-46	1.178e-58	–

Table B.21: p-values for sequential execution time in problem thresh

language	chapel	cilk	go	tbb
chapel	–	1.000e+00	1.000e+00	1.000e+00
cilk	2.179e-44	–	6.336e-62	2.651e-38
go	8.857e-43	1.000e+00	–	1.000e+00
tbb	2.522e-44	1.000e+00	2.041e-72	–

Table B.22: p-values for parallel execution time in problem thresh

language	chapel	cilk	go	tbb
chapel	–	1.000e+00	1.000e+00	1.000e+00
cilk	1.676e-53	–	2.178e-73	3.713e-27
go	1.974e-50	1.000e+00	–	1.000e+00
tbb	1.801e-52	1.000e+00	2.611e-83	–

Table B.23: p-values for sequential execution time in problem winnow

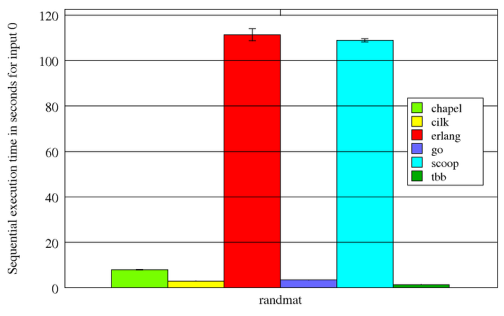
language	chapel	cilk	go	tbb
chapel	–	1.000e+00	1.000e+00	1.000e+00
cilk	4.568e-64	–	1.586e-49	1.000e+00
go	2.141e-82	1.000e+00	–	1.000e+00
tbb	2.389e-66	1.282e-57	1.161e-57	–

Table B.24: p-values for parallel execution time in problem winnow

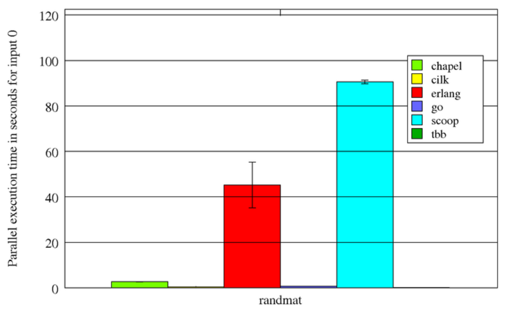
language	chapel	cilk	go	tbb
chapel	–	1.000e+00	1.000e+00	1.000e+00
cilk	2.902e-54	–	9.017e-51	1.000e+00
go	6.822e-50	1.000e+00	–	1.000e+00
tbb	2.551e-51	1.005e-34	3.421e-84	–

APPENDIX C ERLANG AND SCOOP EXECUTION TIME

Figure C.1 shows the execution time for solving the problem Randmat in all languages, including Erlang and SCOOP. The input is the same as in section 5, except for SCOOP, where the input is 10 times smaller (nrows = 2000 instead of 20000). Erlang is more than 10 times slower than Chapel both in the sequential and the parallel versions. SCOOP is 100 times slower than Chapel in the sequential version, and more than 300 times slower in the parallel one.



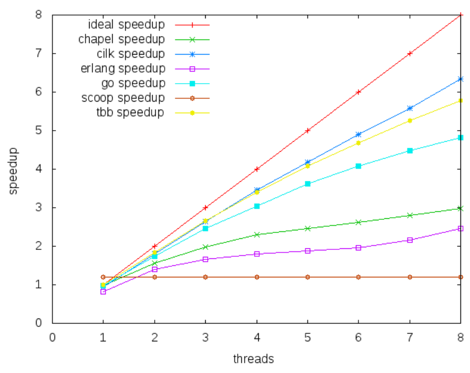
(a) Sequential



(b) Parallel (using 8 threads)

Figure C.1: Execution time (in seconds) for problem Randmat

The speedup for Randmat problem in all languages is shown on figure C.2, for the same inputs mentioned on the previous paragraph. Erlang's speedup almost as good as the Chapel one. Speedup for SCOOP is constant because it is currently not possible to restrict the number of system threads it uses.

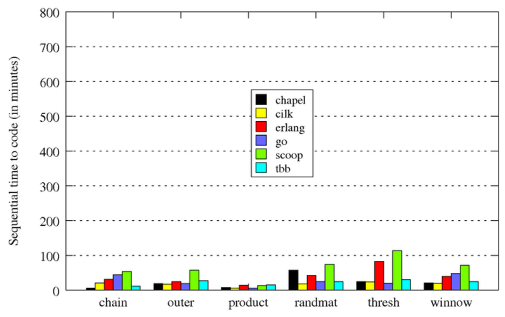


(a) randmat

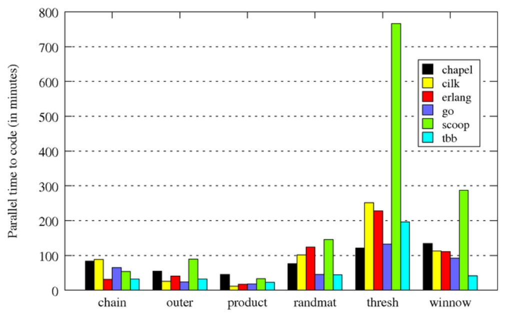
Figure C.2: Speedup for problem Randmat, in all languages

APPENDIX D ADDITIONAL GRAPHS

This chapter shows additional result graphs. Figures D.1, D.2 and D.3 show absolute time to code, number of lines of code and number of words of code, respectively.

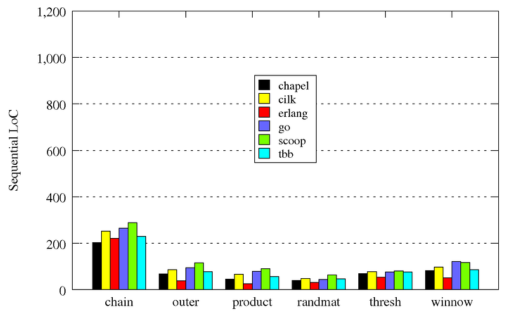


(a) sequential version

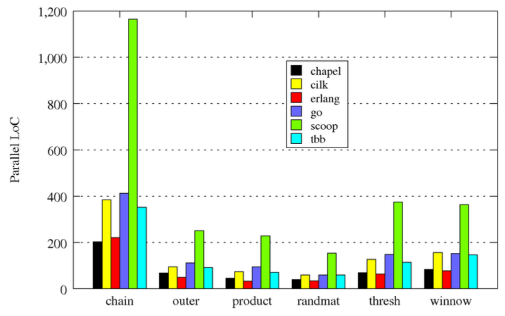


(b) parallel version

Figure D.1: Time to code (in minutes)



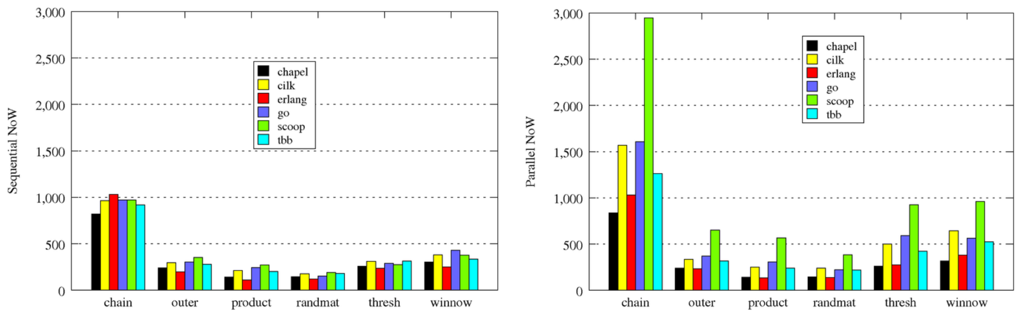
(a) sequential version



(b) parallel version

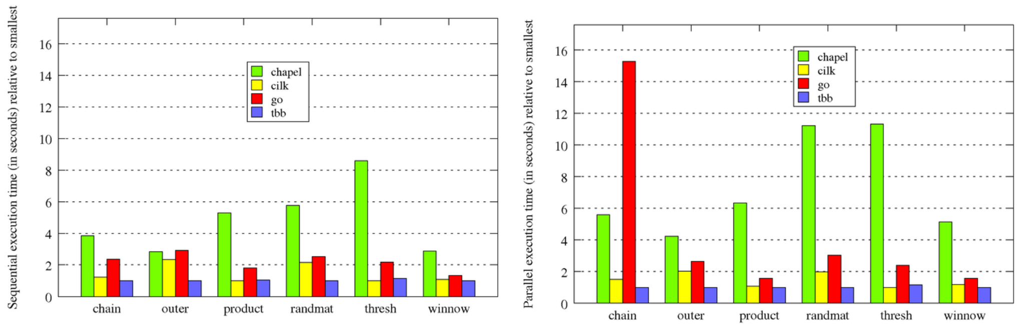
Figure D.2: Number of lines of code

Figure D.4 shows the relative execution time of each language in each problem compared to the fastest execution time on the problem.



(a) sequential version (b) parallel version

Figure D.3: Number of words



(a) Sequential (b) Parallel (using 8 threads)

Figure D.4: Execution time (in seconds) relative to smallest