Semantic Language Extensions for Implicit Parallel Programming

Prakash Prabhu

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science

Advisor: Professor David I. August

September 2013

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
SEP 2013		**00-00-2013 to 00-00-2013**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Semantic Language Extensions for Implicit Parallel Programming	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Princeton University,Princeton,NJ,08544**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Several emerging fields of science and engineering are increasingly characterized by computationally intensive programs. Without parallelization, such programs do not benefit from the increasing core counts available in todays chip multiprocessors. However, writing correct and well-performing parallel programs is widely perceived to be an extremely hard problem. In order to understand the challenges faced by scientific programmers in effectively leveraging parallel computation, this dissertation first presents an in-depth field study of the practice of computational science. Based on the results of the field study, this dissertation proposes two new implicit parallel programming (IPP) solutions. With IPP, artificial constraints imposed by sequential models for automatic parallelization are overcome by use of semantic programming extensions. These preserve the ease of sequential programming and enable multiple parallelism forms without additional parallelism constructs, achieving the best of both automatic and explicit parallelization. The first IPP solution, Commutative Set, generalizes existing notions of semantic commutativity. It allows a programmer to relax execution orders prohibited under a sequential programming model with a high degree of expressiveness. The second IPP solution WeakC, provides language extensions to relax strict consistency requirements of sequential data structures, and dynamically optimizes a parallel configuration of these data structures via a combined compiler-runtime system. This dissertation evaluates both Commutative Set and WeakC on real-world applications running on real hardware, including some that are actively used by some scientists in their day-to-day research. The detailed experimental evaluation results demonstrate the effectiveness of the proposed techniques.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**204**	

© Copyright by Prakash Prabhu, 2013.

All Rights Reserved

Abstract

Several emerging fields of science and engineering are increasingly characterized by computationally intensive programs. Without parallelization, such programs do not benefit from the increasing core counts available in todays chip multiprocessors. However, writing correct and well-performing parallel programs is widely perceived to be an extremely hard problem. In order to understand the challenges faced by scientific programmers in effectively leveraging parallel computation, this dissertation first presents an in-depth field study of the practice of computational science.

Based on the results of the field study, this dissertation proposes two new implicit parallel programming (IPP) solutions. With IPP, artificial constraints imposed by sequential models for automatic parallelization are overcome by use of semantic programming extensions. These preserve the ease of sequential programming and enable multiple parallelism forms without additional parallelism constructs, achieving the best of both automatic and explicit parallelization.

The first IPP solution, Commutative Set, generalizes existing notions of semantic commutativity. It allows a programmer to relax execution orders prohibited under a sequential programming model with a high degree of expressiveness. The second IPP solution, WeakC, provides language extensions to relax strict consistency requirements of sequential data structures, and dynamically optimizes a parallel configuration of these data structures via a combined compiler-runtime system.

This dissertation evaluates both Commutative Set and WeakC on real-world applications running on real hardware, including some that are actively used by some scientists in their day-to-day research. The detailed experimental evaluation results demonstrate the effectiveness of the proposed techniques.

Acknowledgements

First, I would like to thank my advisor, David I. August for everything that has made this dissertation possible. David's passion and vision for research, his ability to identify new and interesting directions for solving hard research problems and his immense faith and courage in relentlessly pursuing these directions successfully for years on end have always been inspiring. His approach of only demanding the best for and from me, though seemed like a bitter medicine at times, has only made me a better student and a researcher in the long run. David's faith in my abilities and his consistent encouragement all throughout my years in graduate school, more so when I was faced with tough situations, have been indispensable to my success as a graduate student. For all this and more – providing me with a platform within the Liberty Research Group, opportunities to work on exciting research projects, proactively encouraging and recommending me at external research avenues, moulding my writing and presentation skills and for providing me with many unique life lessons, I am extremely grateful and thankful.

Next, I would like to thank the rest of the members of my PhD committee. I thank Prof. David Walker and Prof. Jaswinder Pal Singh for reading this dissertation and providing me with interesting feedback which helped improve it in myriad ways. They also suggested significant improvements during my preliminary FPO from which this dissertation has benefited. I would like to also thank Prof. Margaret Martonosi and Prof. Kai Li for serving as my thesis committee members. Their insightful feedback on my work and presentation during my preliminary FPO has been extremely valuable.

I would like to thank members of the Liberty Research group for creating an exceptional research atmosphere, and their support over years ranging from critical reviews of early paper drafts to their support in conducting the field study to last-minute help in writing scripts and experiments before paper deadlines. I thank Tom for the innumerable brainstorming sessions we had as a result of which my research benefited immensely. His engineering

expertise and incredible attention to detail are at least two things that I can only aspire to emulate. I thank Arun for some of most engaging intellectual conversations on the widest range of topics I have ever had with anyone over a period of few years. I thank Nick for his contribution to the compiler infrastructure. His creativity in solving research problems and expertise in architecting and implementing solutions have always been exemplary. I would like to thank Ayal Zaks for his support and the numerous lively discussions we had during his visit at Princeton and also during our later collaborations. Yun is one of the nicest persons I know, and I thank her and Jialu for various discussions held by the window-facing desks of 223. I thank Deep – his excellent analytical thinking coupled with deep interests in several topics have served as the seed for many engaging conversations. Feng's and Taewook's work ethics are the stuff of legend, and I continue to be awed by them. I thank Hanjun and Stephen for giving very insightful feedback on presentations and paper drafts. I would like to acknowledge the support of Matt and Jordan during paper deadlines; Jack and Thomas Mason for their help. I also thank Guilherme and Easwaran for their help during my initial year at Princeton and the many enlightening discussions I had with them.

I would also like to thank the staff of Computer Science Department. I would like to thank Melissa Lawson for simplifying many of the complexities of graduate school and in always accommodating many of my last-minute requests. I thank Google for supporting my research with fellowship support and to the Siebel Scholars program for their support during the later years of graduate school. I would like to acknowledge the support for this work provided by National Science Foundation via Grants 1047879, 0964328, and 0627650, and United States Air Force Contract FA8650-09-C-7918. Additionally, I would like mention that the materials from Chapter 2 and Chapter 4 have been published and presented publicly at SC 2011 and PLDI 2011 respectively.

I would like to thank Prof. Priti Shankar, my Masters advisor at the Indian Institute of Science for introducing me to compilers and program analysis research. Her warm and

compassionate words filled with genuine concern for all her students is something I will always cherish. I would also like to thank Prof. Govindarajan, my academic advisor at IISc, for his advice and initial tips on critical reading of research papers. I would like to thank my internship mentors, Kapil Vaswani, Ganesan Ramalingam, Gogul Balakrishnan, Franjo Ivancic, Naoto Maeda and Aarti Gupta for providing me with valuable and enjoyable research experience in an industry setting. In particular, I would like to thank Kapil and Franjo for recommending me, long after my internships got over.

Outside of work, I am deeply appreciative all the enriching conversations and time spent with the wonderful people I've met and am friends with. I thank Easwaran for all his guidance on grad school life during my initial years at Princeton – everything from the joint cooking sessions to regular PHS meetings were highly memorable. Tushar, thanks for all the time spent together during the first two years and the in-depth phone conversations ever since – your meticulous approach towards so many things have always been inspiring. Arun, thanks for being an exemplar of grad-school-resilience and for all the shared time – ranging from forays into testing out "healthy" diet regimens to playing basketball to occasional parsing of verses from *Vivekachudamani*. Ajay, thanks for being my roommate for 3+ years – your foresight and zeal for pursuing new ideas have always been refreshing. I would also like to acknowledge Arnab, Anuradha and Divjyot for their support.

Stimit, thanks for all the stimulating discussions on everything from nature to nurture, the joint work sessions at Frist/Small World/Starbucks and squash games. Your competitive spirit is amazing, I probably learnt a thing or two about competing from you. Thanks Deep for your exemplifying the hit-the-round-running attitude, for epitomizing passion for football and not to forget your amazing penchant for writing. Srinivas Ganapathy Narayana (who will not have his name specified any other way), thanks for the myriad long and in-depth conversations on so many topics of shared interests, your rational and approach

to problem solving always shines through. The three of you made my last two years at Princeton really fun and memorable.

I thank Sriram for being a great and affable friend – being there to make me feel at home when I arrived in the US, the great many conversations on virtually any topic under the sun, your passion for history, the time spent at Princeton, Toronto and REC, the list goes on. I thank Raghavan for being a role model in so many respects – the uber-energetic passion for everything you set your mind on and your happy-go-lucky attitude towards many things have always been influential for people around you. Arvind has never ceased to amaze with his zeal for learning new things all the time, and our discussions on static analysis at IISc were instrumental in deepening my interest in pursuing research. Rajdeep's energy and his cheerful optimism are highly infectious. I thank Mani and Kaushik for various grad-school related discussions on wide-ranging topics from paper review and conference related things to systems design and implementation ideas throughout my PhD. I would also like to acknowledge Vishal and Rajesh. I thank Naveen, Kannan, Subba, Dilip and the rest of "decagonites" for all the wonderful time spent together at REC, Hyderabad and at RRI.

I cannot thank my family enough for their love, affection and support which has made me who I am today. I am indebted to Amma, Aanu, Bapama, Smitha, Anna, Vanni, Sangeetha akka and little Aditi for being there for me at all times rough and smooth and for making all the hard work worth it. Amma and Aanu have overcome innumerable hardships to make sure we get a good education and have a great environment for learning, and have always encouraged and supported us to freely pursue our interests. Amma's care and love, her meticulousness and her wish for the best for her children, Aanu's dedication to learning and his calm, reassuring and loving concern for us have always stood us in good stead. Bapama's love for her grandchildren, and her strength in the face of adversity will forever be inspiring for me. Smitha's love for all of us is special. Anna has always been full of ideas

since our early childhood, I've sought to imitate him in so many respects, and I am so fortunate to benefit from his forthright and clear views and advice on so many things. Vanni's simplicity, her courage and entrepreneurial spirit and smiling demeanor have always been amazing. Aditi's smile and enthusiasm have overpowering charm and a few minutes with her are enough to enliven even the dreariest of proceedings. Sangeetha akka never ceases to inspire me with her strength and views and her clear and rational advice on everything has been crucial for helping me complete this dissertation – thank you and Niyant for helping me in times of stress, and by hosting me so many times at NYC with cheerful enthusiasm. I would also like to thank Mamama, Prasad maam, Geetha mai, Nitish, Goplee, Jyotsna, all my Mhavs, Manthus, and other cousins for their love and affection.

Contents

List of Figures

Chapter 1

Introduction

Computational science [182], a multidisciplinary field encompassing various aspects of science, engineering, and computational mathematics is increasingly being seen as the "third approach" [66], after theory and experiment, to answering fundamental scientific questions. Researchers practicing computational science typically face two concerns competing for their time. Primarily, they must concentrate on their scientific problem by forming hypotheses, developing and evaluating models, performing experiments, and collecting data. At the same time, they also have to spend considerable time converting their models into programs and testing, debugging, and optimizing those programs.

In the past two decades, there has been an exponential increase in the amount of data generated and computation performed within many scientific disciplines [182, 194], signifying an increasing need for high performance computing. A particularly stark example is in the field of Computational Biology. Figure 1.1 shows the growth in the number of recorded DNA base pairs and sequences within GenBank [31], a genetic database that has an annotated collection of all publicly available DNA sequences. As shown by the graph, the number of bases in GenBank has doubled approximately every 18 months in the past

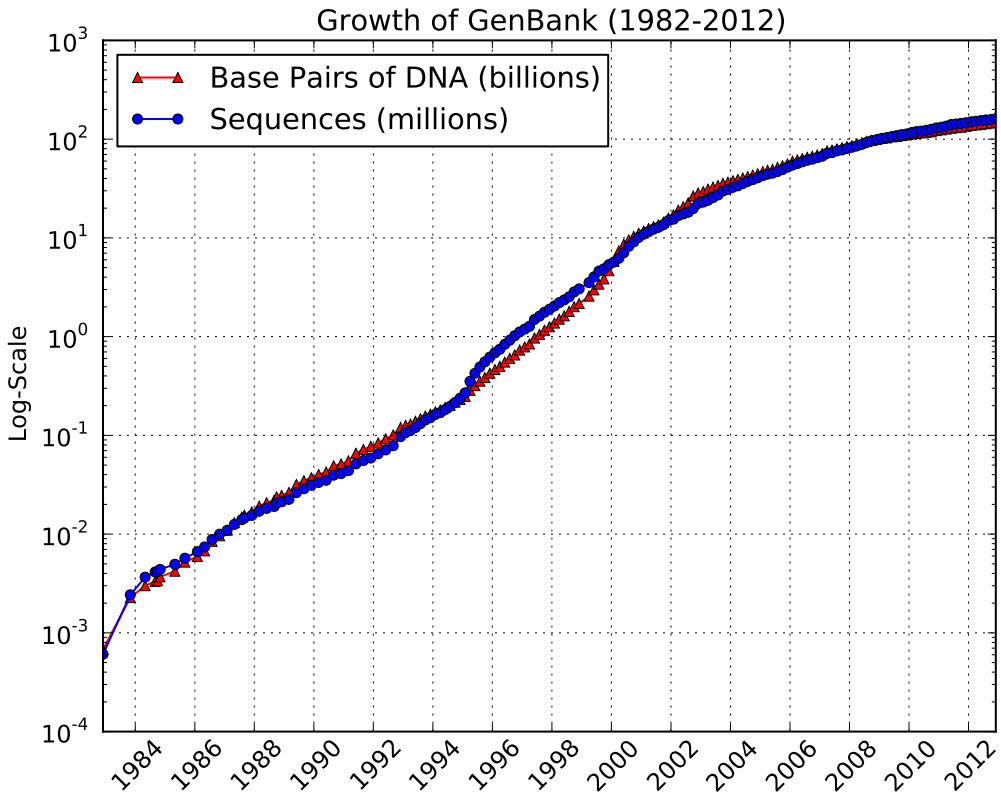


Figure 1.1: Plot showing the growth in the number of recorded DNA base pairs and sequences within GenBank, in log-scale, over the last two decades. GenBank [31] is a genetic sequence database that has an annotated collection of all publicly available DNA sequences. Data for the plot was obtained from the GenBank webpage [11].

two decades. Given this data growth rate, the need for high performance computing for retrieving, processing, and analyzing this data is more important now than ever.

Concurrently, the semiconductor industry is undergoing a paradigm shift in the design of newer generations of microprocessors, signifying the importance of parallelism for both small and large scale computing. Figure 1.2 shows the performance of various commodity processors from 1992 to 2013 using the SPEC [189] benchmark suite. Up until 2004, improvements in hardware and innovations in microarchitecture had guaranteed an exponential performance growth for single-threaded programs without any software changes. However, this rate of growth has decreased substantially since 2004 due to power and

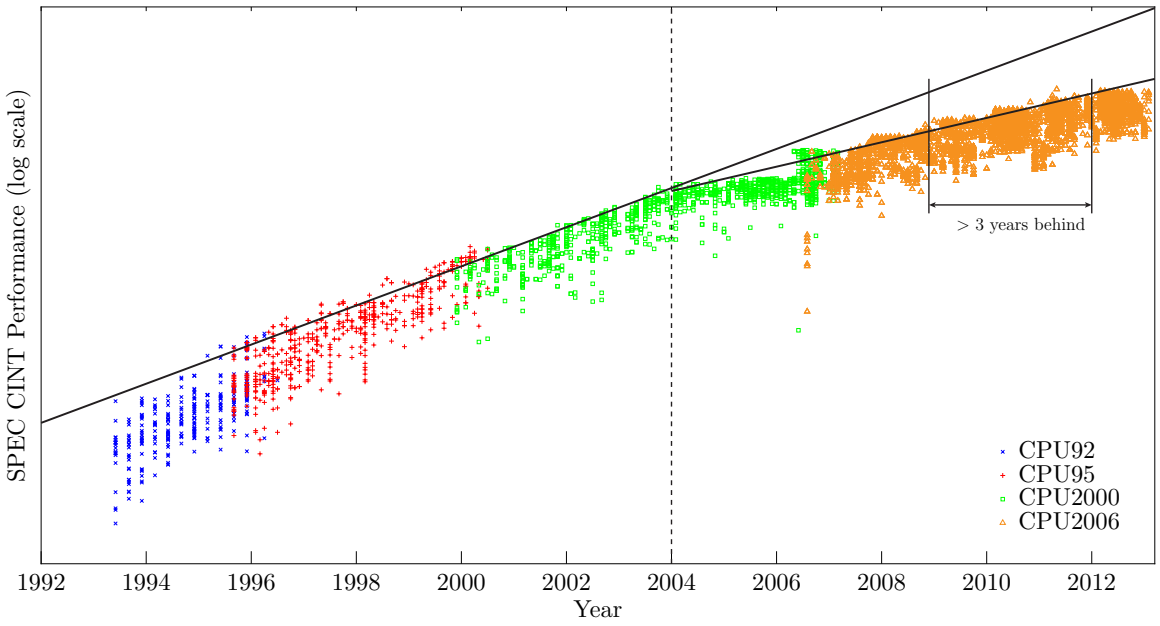


Figure 1.2: Normalized SPEC scores for all reported configuration of machines between 1992 and 2013 [136].

thermal walls encountered when scaling clock frequency and due to the increased design complexity of aggressive out-of-order processing substrates aimed at exploiting instruction level parallelism. As a result, this has forced a paradigm change in the design of processors, with the leading manufacturers resorting to packing the exponentially increasing number of transistors resulting from Moore's law [149] into a chip multiprocessor. These single-chip multiprocessors, or multicore have multiple independent processing cores packed on a single die. Consequently, in the multicore era, parallel hardware is ubiquitous and the only way to sustain the decades old performance trends is creating parallel programs that allow simultaneous execution of multiple threads of control on the different cores.

Conventional wisdom has held that writing correct and well-performing parallel programs is an extremely tedious and error-prone task [110]. However, to date, there has been no in-depth formal study of the problems faced by scientific programmers in effectively leveraging parallel computation in their disciplines. To this end, this dissertation presents an *in-depth* study of the practice of computational science at Princeton University,

a RU/VH institution, which stands for "very high research activity doctoral-granting university" as classified by the Carnegie Foundation [7]. This study was conducted through a field study of researchers from diverse scientific disciplines and covers important aspects of computational science including parallel programming practices commonly employed by researchers, the importance of computational power, and performance enhancing strategies in use. This field study was conducted through personal interviews with 114 researchers randomly selected from diverse scientific fields.

The analysis of results from the field study reveals several interesting patterns. In contrast to the popular view that scientists use only numerical algorithms written in MATLAB and FORTRAN, the field study discovered that C, C++, and Python were popular among many scientists and there is a growing need for non-numerical algorithms. Interestingly, a substantial portion of scientific computation is sequential and still takes place on scientists' desktops, most of which are multicore. In spite of this, knowledge of shared-memory parallelization techniques in the scientific community is low. Furthermore, the field study reveals that the dominant perception of parallel programming within the community is that of black art. Some of the reactions to the difficulty of parallel programming were of the form "it is hard", "big learning curve", "not a good experience", and "implementation time too high".

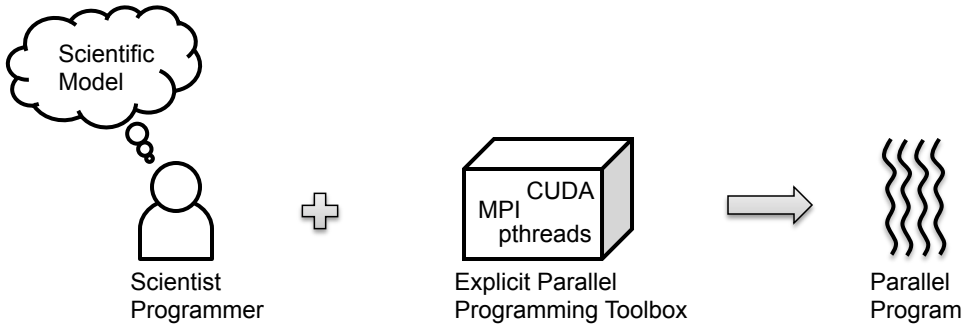
Based on the results of the field study, this dissertation adopts the approach of implicit parallel programming (IPP) and contributes two new solutions within this framework. In order to see the benefits of IPP, consider two very different alternatives: explicit parallel programming and automatic parallelization.

1.1 Approaches to Obtaining Parallelism

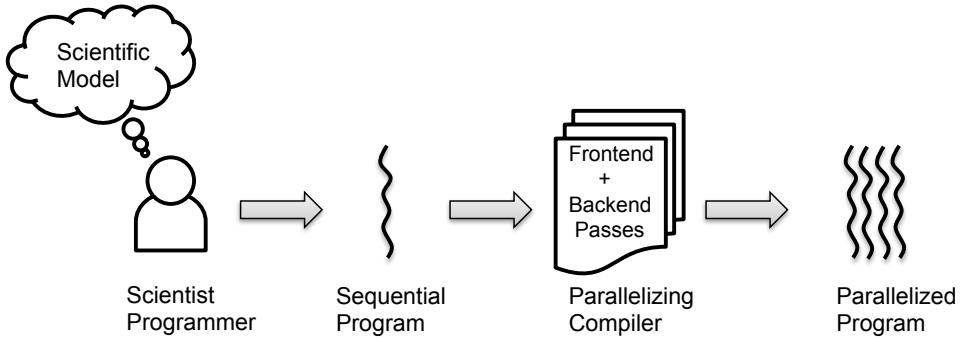
Explicit parallel programming [43, 46, 218] is the dominant parallel programming model for multicore today. These models require programmers to expend enormous effort reasoning about complex thread interleavings, low-level concurrency control mechanisms, and parallel programming pitfalls such as races, deadlocks, and livelocks. Despite this effort, manual concurrency control and a fixed choice of parallelization strategy often result in parallel programs with poor performance portability. Consequently, parallel programs often have to be extensively modified when the underlying parallel substrates evolve, thus breaking abstraction boundaries between software and hardware.

Recent advances in automatic thread extraction [173, 205, 220] present a promising alternative. They avoid pitfalls associated with explicit parallel programming models but retain the ease of reasoning of a sequential programming model. However, sequential languages often express a more restrictive execution order than required by various high-level algorithm specifications. As a result, automatic parallelizing tools, overly constrained by the need to respect the sequential semantics of programs written in languages like C/C++, are unable to extract scalable performance.

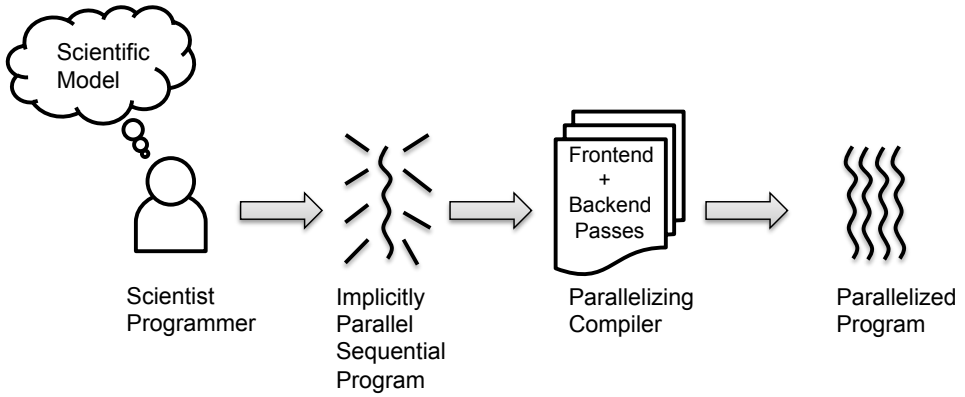
Implicit parallel programming models (IPP) [34, 95, 104, 208] offer the best of both approaches. In such models, programmers implicitly expose parallelism inherent in their program without the explicit use of low-level parallelization constructs. An interesting subclass of models within this space includes those that are based on top of sequential programming models [104]. In such models, programmer insights about high-level semantic properties of the program are expressed via the use of extensions to the sequential model. Transformation tools then exploit these extensions to automatically synthesize a correct parallel program. This approach not only frees the programmer from the burden



(a) Explicit Parallel Programming



(b) Automatic Parallelization



(c) Implicit Parallel Programming

Figure 1.3: Workflow of explicit and automatic parallelization compared with implicit parallel programming

of having to worry about the low-level details related to parallelization, but also promotes retargetability of such programs when presented with newer parallel substrates.

Figure 1.3 compares the explicit parallel programming and automatic parallelization with implicit parallel programming within the context of scientific computing. In a scenario

typical in many scientific disciplines, a scientist-programmer starts with a scientific model at hand and wants to convert this model into an efficient and scalable parallel program.

With explicit parallel programming approaches (Figure 1.3(a)), the abstraction boundaries between hardware and software are weak. A scientist programmer has typically learn a new parallel programming model for every new architecture type. Although such programming models are generally usable for a few successive generations of such architectures, scientists programmers typically spend enormous programming effort in performance debugging for every new generation. For instance, `pthreads` [43] based programs that run on shared memory machines have to be manually modified using message passing libraries like `MPI` [150] to run on clusters. These models force the learning of low-level concurrency control mechanisms that still does not ensure performance portability. However, since a programmer has fine-grained control over parallel execution, programmer insights about high level semantic properties can be explicitly encoded in these programs to a great amount of detail.

With automatic parallelization (Figure 1.3(b)), the scientific model is encoded into a sequential program, which scientists are more comfortable in doing than writing explicit parallel programs. This program is parallelized with the help of parallelizing compiler. This approach retains the ease of sequential programming and does not break the abstraction boundaries between software and hardware. However, in writing the program in a sequential programming model, the programmer insight about high level algorithmic properties about the scientific model is filtered out, which limits the freedom given to the parallelizing compiler for parallelization.

Implicit parallel programming (Figure 1.3(c)) represents an approach that combines the best of both explicit parallel programming and automatic parallelization. Here, the scientist still encodes her model in the form of a sequential program as earlier, but insights

about the semantic properties of the model are expressed via annotations that relax sequential semantics by introducing controlled levels of non-determinism within the sequential program. This approach still retains the ease of sequential programming, while providing additional freedom to the parallelizing compiler to obtain a much better parallel schedule than before.

1.2 Dissertation Contributions

This dissertation presents two new semantic programming extensions and their associated implicit parallel programming solutions. The first solution, called Commutative Set [165] or COMMSET, generalizes existing semantic commutativity constructs and enables multiple forms of parallelism from the same specification. COMMSET has several advantages over existing semantic commutativity constructs: it allows commutativity assertions on both interface declarations and in client code, syntactically succinct commutativity assertions between a group of functions, and enables multiple parallelism forms without additional parallelism constructs.

The second IPP solution, WEAKC, is based on a data-centric language extension that is used to weaken the semantics of sequential data structures that are commonly employed in several classes of combinatorial search and optimization algorithms. Accesses to these data structures, while crucial to improving the convergence times of the main search or optimization loops, create dependences that obstruct parallelization. Weakening the semantics of these data structures can expose new parallelism opportunities without violating programmer intentions. WEAKC provides language extensions to express this weakened semantics, and dynamically optimizes a parallel configuration of these sequential data structures via a runtime system.

Both COMMSET and WEAKC have been implemented as extensions of the clang/LLVM [130] compiler framework, and are evaluated on several real-world applications running on real hardware. These applications include programs that were collected during the field study which are used by some scientists in their day-to-day research. Based on experimental results, this dissertation demonstrates the effectiveness of the proposed techniques.

To summarize, this dissertation has the following contributions:

- An in-depth field study of the practice of computational science at a RU/VH institution that was conducted through personal interviews with 114 researchers randomly selected from diverse fields of science and engineering, and an analysis of results from the field study that reveals several interesting patterns that motivate the IPP approach.

- The design, syntax, and semantics of COMMSET, a novel programming extension that generalizes, in a syntactically succinct form, various existing notions of semantic commutativity; and an end-to-end implementation of COMMSET within a parallelizing compiler that enables multiple parallelizations with automatic concurrency control.

- A semantic parallelization framework, WEAKC, that provides novel programming extensions for weakening the consistency of sequential data structures to implicitly expose parallelism in loops using them; and a compiler and a runtime system that dynamically optimizes a parallel configuration of the WEAKC data structures.

- A demonstration of the effectiveness of COMMSET and WEAKC in extending several real-world applications with a modest number of annotations that require minimal programmer effort for effectively parallelizing them on real hardware. These applications include programs collected during the field study, which are used by scientists in their day-to-day research.

1.3 Dissertation Organization

This dissertation is organized as follows. Chapter 2 presents a field study of the practice of computational science conducted within a research university. Chapter 3 describes the motivation behind the adoption of implicit parallel programming (IPP), and compares IPP with different paradigms for parallel programming and parallelization. Chapter 4 describes the limitations of prior semantic commutativity proposals and the design, syntax, semantics and implementation of COMMSET. Chapter 5 describes the motivation behind the design of weakly consistent data structures and the syntax, semantics and implementation of WEAKC. Chapter 6 presents an experimental evaluation of the techniques proposed in this dissertation. Chapter 7 presents an in-depth discussion of existing work related to the proposals in this dissertation. Chapter 8 discusses promising areas for future research and concludes.

Chapter 2

Field Study

This chapter presents a field study of scientists from diverse disciplines, practicing computational science at a doctoral-granting university with exceptionally high research activity. The field study covers many things, among them, prevalent programming practices within this scientific community, the importance of computational power in different fields, use of tools to enhance performance and software productivity, computational resources leveraged, and prevalence of parallel computation. The results motivate the implicit parallel programming techniques presented in this dissertation, and at the same time reveal several patterns that suggest interesting avenues to bridge the gap between scientific researchers and programming tools developers.

2.1 Methodology

The field study covers a set of 114 randomly selected researchers from diverse fields of science and engineering at Princeton University. The pool of field study candidates includes all graduate students, post doctoral associates, and research staff in various scientific disciplines at Princeton University. An email soliciting participation in the field study was

Field	Discipline	Count
Natural Sciences	Astrophysics	3
	Atmospheric and Oceanic Sciences	2
	Chemistry	5
	Ecology and Evolutionary Biology	5
	Physics	5
	Geosciences	6
	Molecular Biology	4
	Plasma Physics	2
Engineering	Chemical	7
	Civil and Environmental	5
	Mechanical and Aerospace	11
	Electrical	12
	Operations Research and Financial	5
Interdisciplinary Sciences	Music	4
	Applied and Computational Math	2
	Computational Biology	4
	Neuroscience/Psychology	13
Social Sciences	Economics	10
	Sociology	5
	Politics	4
Total		114

Table 2.1: Subject population distribution

initially sent to randomly selected candidates from the university database. The email mentioned "use of computation in research" as a criterion for participation. After a candidate had replied indicating interest in the field study, an interview was conducted exploring, in depth, the various aspects of scientific computing related to the candidate's research.

Table 2.1 shows the distribution of subjects across different scientific fields. In this dissertation, the word "scientist" is used in a broad sense, to cover researchers from natural sciences, engineering, interdisciplinary sciences, and social sciences. A total of 20 disciplines were represented. Of the 114 subjects, 32 were from the natural sciences, 40 from engineering, 23 from interdisciplinary sciences and 19 from the social sciences. Most of the interviewees were graduate students in different stages of their research. Six interviewees were postdoctoral researchers and research staff. Barring two instances, researchers from the same discipline were from different research groups.

The field study was conducted through personal interviews, in order to allow for a deeper understanding of the different computing scenarios and situations unique to each subject. Each interview conducted was in the form of a discussion that lasted for about 45

minutes. All the interviews were conducted over a period of 8 months. The field study covered three main themes central to scientific computing: (a) programming practices (b) computational time and resource use, and (c) performance enhancing methods.

2.2 Results

This section presents the results of the field study. To begin with, the scientific computing environment at Princeton University is characterized. With this as the background, the results of the field study are presented suitably categorized into the three themes mentioned above. Each theme is introduced by posing a broad set of questions, and then answering these questions through a general set of patterns observed during the field study along with data to substantiate each observation. To highlight these key patterns, and other central ideas or conclusions that appear later in this chapter we set them apart from the main text as a subsection heading in bold.

2.2.1 Computing Environment

Researchers at Princeton University are heavily supported in terms of computational resources and expertise. The Princeton Institute for Computational Science and Engineering (PICSciE) [9] aims to foster the computational sciences by providing computational resources as well as the experience necessary to capitalize on those resources. At the time of writing, these resources include the larger cluster hardware available through the Terascale Infrastructure for Groundbreaking Research in Engineering and Science (TIGRESS) [6]. TIGRESS is a high performance computing center that is an outcome of collaboration between PICSciE, various research centers [4, 5, 8, 10], and a number of academic departments and faculty members.

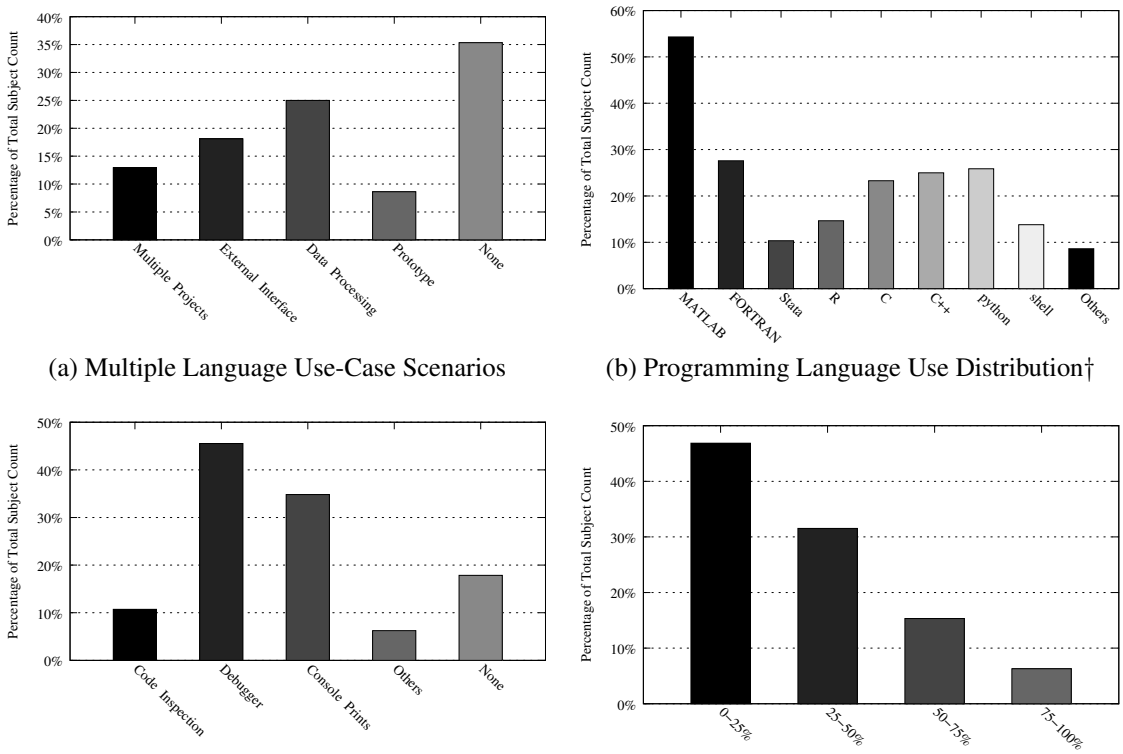
TIGRESS offers four Beowulf clusters (with 768, 768, 1024, and 3584 processors), and a 192 processor NUMA machine with shared memory and 1 petabyte of network attached storage. These clusters serve the computational needs of 192 researchers. Administrators at TIGRESS estimate that their systems are at 80% utilization. Additionally, PICSciE offers courses, seminars and colloquia to aid the computational sciences. Since 2003, PICSciE has offered mini-courses on data visualization, scientific programming in Python, FORTRAN, MATLAB, Maple, Perl and other languages, technologies for parallel computing (including MPI and OpenMP), as well as courses on optimization and debugging parallel programs. Recently, PICSciE began offering a course on scientific computing. PICSciE also offers programming support for troubleshooting malfunctioning programs, parallelizing existing serial codes, and tuning software for maximum performance.

2.2.2 Programming Practices

Representative questions concerning this theme included: What kind of programming paradigms, languages and tools do scientists use to perform scientific computation? Do scientists employ effective testing methods to certify the results of their programs? What fraction of research time do scientists spend in programming or developing software?

Scientists commonly interface a diverse combination of numerical, general purpose, and scripting languages.

One common perception is that scientific computation is dominated by heavy array-based computation in languages that are specially tuned for numeric computation like FORTRAN and MATLAB [148, 167]. Contrary to this perception, our field study indicates that scientists commonly interface a diverse combination of numerical, general purpose and scripting languages in an ad-hoc manner. Around 65% of scientists use at least one combination of numerical/scripting language and a general purpose language.



(a) Multiple Language Use-Case Scenarios

(b) Programming Language Use Distribution†

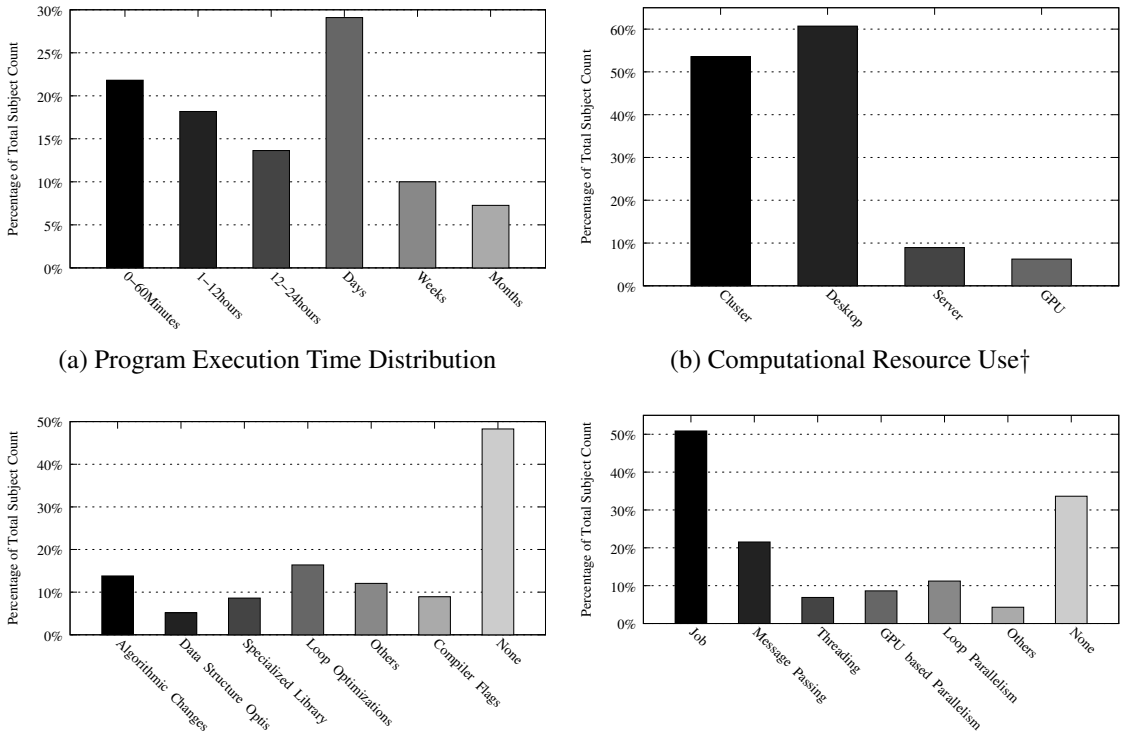
(c) Debugging Techniques Employed

(d) Programming Time, as Percentage of Research Time

Figure 2.1: Survey Data I. The categories in the graphs marked † are not mutually exclusive and do not sum to 100%.

The distribution of multiple-language use-case scenarios is shown in Figure 2.1(a). Close to one-fourth of scientists used numerical computing/scripting languages only for pre- and post- processing of simulation data, while writing all of the heavy duty computation code in general purpose languages like C and C++ for performance.

18% of scientists leveraged external interface functionality provided by languages like MATLAB to call into libraries pre-written in C/C++. Interfacing of this kind often involved writing wrappers to emulate the native programming model. For instance, researchers in Ecology and Evolutionary Biology, wrote a "call by reference" emulation framework in MATLAB when interfacing pointer-based data acquisition code written in C.



(a) Program Execution Time Distribution

(b) Computational Resource Use†

(c) Distribution of Performance enhancing strategies†

(d) Use of Parallelism†

Figure 2.2: Survey Data II. The categories in the graphs marked † are not mutually exclusive and do not sum to 100%.

Nearly 9% of researchers used scripting/numerical languages for fast prototyping. They wrote their programs first in MATLAB/Python-like languages and tested their algorithms on small data sets. Once tested, these programs were re-written in C and C++ for bigger data sets. Around 13% of researchers used both numerical and general purpose languages for different projects, typically influenced by the perceived productivity versus performance trade-offs for the given problem at hand.

The distribution of different kinds of programming languages employed by scientists in the field study is shown in Figures 2.1(b) and 2.3. As Figure 2.3 shows, there is considerable overlap between the use of general purpose, scripting and numerical languages. The most dominant numerical computing language in use was MATLAB – more than half

the researchers programmed with it. This is followed by FORTRAN, which was used by around 27% of researchers. Researchers using FORTRAN were influenced by the availability of legacy code, typically written by their advisors during the early nineties. More than 40% of surveyees relied on a general purpose language to deliver on computational performance. Two specific languages, C and C++, dominated in equal measure. An interesting point was the discipline-wise stratification of researchers using C/C++ and FORTRAN code. Researchers working in emerging interdisciplinary fields like Psychology, Neuroscience and Computational Biology wrote programs in C/C++. By contrast, most of the FORTRAN use was restricted to established scientific fields in natural sciences like Astrophysics, Chemistry, and Geosciences. The dominant scripting language in use was Python. Around one-fourth of interviewees used Python, with shell scripts as the second favorite. Apart from normal string and data parsing and processing, researchers leveraged several scientific packages written in Python like SciPy [111], NumPy [156], and Biopython [63].

In the fields of sociology, politics, music, and astrophysics, the use of domain specific languages (DSLs) was nearly universal. Although DSLs are not as general as other programming solutions, their higher level specialized programming constructs are best suited for tasks that are repeated sufficiently often within each scientific domain. Interactive Data Language (IDL) [2] is an example of a DSL that is hugely popular amongst astrophysicists. BUGS [164], a DSL for describing graphical models, is almost the *de facto* standard amongst researchers studying politics. Chuck [209] and Max/MSP [3] are DSLs that are heavily used by researchers in Music for audio and video processing.

Scientists spend a substantial amount of research time programming.

On average, scientists estimate that 35% of their research time is spent in programming/developing software. The distribution is shown in Figure 2.1(d). While initially some time is spent on writing code afresh, a considerable portion of time is spent in many tedious

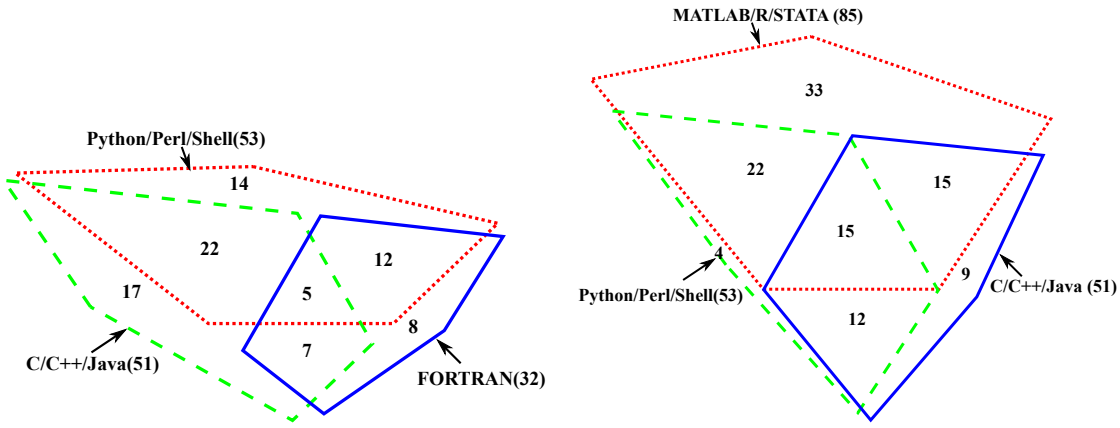


Figure 2.3: Proportional representation of two classes of numerical languages, intersecting with general purpose and scripting languages.

activities. For example, researchers in Politics and Sociology who used R/Stata had to do considerable programming to retrofit census data into formats that individual packages in R/Stata understood. Some researchers in Chemical Engineering had to reverse engineer undocumented legacy code that performed flame simulation, long after the original authors had graduated, in order to adapt the code to newer fuels.

Many researchers also re-wrote code for performing similar tasks rather than templatize and re-use code. Quite often, this was done via a copy, paste and modify procedure which is highly error prone. None of these activities were well tested. Many scientists write programs to solve infrastructure problems unrelated to their research. Some scientists had to write utilities to translate between file-formats expected by different tools. For example, many geophysics applications each use a different data format for representing the model of the earth, making them difficult to interact with each other.

Despite this, a vast majority of these researchers felt that they "spend more time programming than they should," and that research time was better spent in focusing on scientific theories or on experiments ("more concerned about physics," said one researcher). For many researchers, it is often a case of "What is limiting us is not processor speeds but knowledge about programming the machines".

Scientists do not rigorously test their programs.

Given that the computational method has taken over as the method of first choice in many scientific fields [194], one would expect scientists to rigorously test their programs using state-of-the-art software testing techniques. However, our field study results point to the contrary. We asked researchers to estimate the amount of their programming time that they spend on testing and debugging. Although researchers estimate that they spend anywhere from between 50% to 60% of their programming time on finding and fixing errors in their programs, the debugging and testing methods employed were primitive. Only one researcher considered the use of assertions at all in her code. Only three researchers wrote unit tests. Not many researchers were aware of version control systems, given their utility in detecting sources of regressions.

More than half of the researchers did not use any debugger (Figure 2.1(c)) to detect and correct errors in their code. 18% of researchers never tested their programs once they were written, either because they thought it was "too simple" or relied on code written by others, which was assumed to be "well-tested." 11% of researchers relied on "trial and error" to detect bugs, which typically involved checking subsections of code for correctness by commenting out the rest of the code. Given the inherent combinatorial space of code changes that are possible, this process itself was error-prone and slow. A small minority of researchers relied on the expertise of their more knowledgeable peers to fix errors ("call up people who have experience," said one researcher).

Although some scientists acknowledged the gap between theory and practice of computational science, the current outlook of scientists on software and programming in general is not good ("scientists are not interested in software as a first-order concern," as noted by one researcher).

Responses from the field study indicate that software frameworks for scientific computation allow researchers who are just getting started on research to learn faster. Scientists

Scientific Discipline	Application	Software Engineeering					Programming Model/ Language	Typical Execution Scenario	
		Ver	Mail	Doc	Bug	EI		Run time	Architecture(s)
Astrophysics	Athena [195]	✓	-	Extensive	✓	-	MPI + FORTRAN/C	20 hours	Cluster with 192 nodes, 2 cores per node
Neuroscience / Psychology	AFNI [58]	-	✓	Moderate	✓	✓	OpenMP + C	15 hours	Cluster with 52 nodes, 8 cores per node
Chemical Engineering	LAMMPS [163]	✓	✓	Moderate	✓	✓	MPI + C	3 days	Cluster with 4 nodes, 4 cores each
Chemistry	Quantum Espresso [83]	✓	✓	Moderate	-	-	MPI + FORTRAN/C	1 month	Cluster with 384 nodes, 2/4 cores per node
Civil and Environmental Engineering	VIC [134]	-	✓	Moderate	-	-	Sequential + C	1.5 weeks	Cluster with 384 nodes, 2.4 cores per node
Computational Biology	Sleipnir [103]	✓	-	Extensive	-	✓	Pthreads + C++	1 day	Cluster with 58 nodes, 8 cores per node
Economics	Dynare [113]	-	-	Extensive	✓	✓	Sequential + MATLAB/ C++	12 hours	Desktop with 2 cores 2 cores
Electrical Engineering	fftw [80]	-	-	Moderate	-	✓	MPI/Pthreads + C	30 mins	Desktop and GPU
Geosciences	SPECFEM3d [122]	✓	✓	Extensive	✓	-	MPI/Pthreads + FORTRAN/C	30mins to 9hrs	Cluster with 384 nodes, 8 cores per node
Molecular Biology	HMMER [71]	-	-	Moderate	-	✓	Pthreads + C/C++	1 week	Cluster with 58 nodes, 8 cores per node
Physics	GEANT4 [15]	-	-	Extensive	✓	✓	Sequential + C	1 week	Cluster with 70 nodes, 2 cores per node
Politics	JAGS [164]	✓	-	Moderate	✓	✓	Sequential + C++	4 days	Desktop with 2 cores

Table 2.2: Open source compute-intensive scientific applications collected during the field study, along with typical execution scenarios for many interviewees. Software Engineering category notes the use of Version control systems, Mailing Lists, Documentation, Bug tracking, and Extensible Interfaces.

interviewed during the field study used frameworks such as Sleipnir [103], AFNI [58], Quantum Espresso [83], and Dynare [113]. These domain specific scientific coding frameworks provide a variety of builtin features that enable easier implementation and testing of new ideas.

Given the interdisciplinary expertise needed for writing robust scientific code, it is desirable to have many theses focused on scientific tools. Unfortunately, scientists are not rewarded for developing and releasing robust scientific software. As noted by two prominent scientists during the field study, faculty generally believe that development of software tools "does not make for a scientific contribution." Similar sentiments echoed included "If you are not going to get tenure for writing software, why do it?" Alarmingly, even "funding agencies think software development is free," and regard development of robust scientific code as "second class" compared to other scientific achievements.

A few scientific programs conform to best software engineering practices and have high standards of reproducibility.

A recent article that appeared in Nature News [145] cast a rather bleak picture of scientific programming practices, citing complete absence of known software engineering methods, testing, and validation procedures in most scientific computing projects. While our field study results do agree in large part with those results, we also found many interviewees leveraged a small set of open source scientific programs that stood out both in terms of the best software engineering practices and high standards of reproducibility. Typically, each discipline had a few open source programs developed by scientific teams world-wide that were popular and were utilized by many others in their field. In our subject population, around 48% of interviewees used or modified these open source software at some stage of their research.

Table 2.2 lists some of the representative open source programs in each field that are in this category. Most of these programs are results of projects characterized by sophisticated software engineering practices and evolved with goals of reproducibility, validation, and

extensibility. These programs were written by researchers in science rather than professional programmers. These projects are distinguished from the various scientific programs developed in-house by the following critical features:

- **Focus on Extensibility.** Most of these open source programs had extensibility and interoperability as a first order design concern. This often meant that the codes were modular and portable. For instance, AFNI [58] has a plug-in based architecture. AFNI users can write their own plug-ins for analysis and visualization of MRI data. GEANT4 [15] developers specifically adopted object oriented paradigms to allow for customized implementation of several generic interfaces. JAGS [164], although written in C++, has well defined R interfaces. The developers of LAMMPS [163] chose portability over optimality by opting not to use vendor specific APIs for message passing.

- **Long History of Software Development.** The level of sophistication achieved by these programs is the result of years of dedicated programming effort. Often, these programs were in development since the 1990s and underwent significant changes that often made the newer versions more general and portable than the earlier designs. In many cases, the open source program was an outgrowth of collaboration between scientists across different institutions. For instance, SPECFEM3d [122] was initially prototyped for the Thinking Machine using High Performance FORTRAN, and later redesigned using MPI to make it portable to run on different clusters. JAGS was written to achieve a platform independent implementation of an existing Bayesian analyzer. GEANT4, Quantum Espresso [83] and SPECFEM3d were designed and developed over many years by international teams with common interests, and with an experience of having developed similar programs in the past.

- **Performance and "Separation of Concerns."** The current set of open source programs adopt diverse performance optimization techniques. At the same time, the sections of the program that included machine specific optimizations were carefully separated from the machine independent portions of code using modularized interfaces. Many of these programs provide an interface to the user to choose accuracy-performance tradeoffs, often as a compile time or a run time option. Developers of these programs did not make any compromises on performance for providing flexible interfaces. For instance, fftw [80] provides a module called the "planner" that is responsible for choosing the best optimization plan for a given architecture without requiring any additional code changes from the user. SPECFEM3d's flexibility has given rise to versions that are being used across 150 different academic institutions and at the same time has also won the Gordon Bell prize [1, 123] for scalable performance.

- **Extensive testing and validation.** The programs listed in Table 2.2 have a large user base. These programs had extensive testing and validation methods in place. Very often, these open source projects had robust testing frameworks and used version control systems. These programs were accompanied by considerable documentation in the form of tutorials and user guides that made it easier to detect discrepancies in the program behavior. An active user community around the software existed that regularly reported bugs which were quickly fixed. Some of the projects even had a bug tracking system.

2.2.3 Computational Time and Resource Use

Representative questions concerning this theme included: How long do scientists wait for a computer to complete project runs? What kind of computational resources do scientists

typically use to meet their computational needs? How would their research change with faster computation?

Programs run by scientists take on the order of days to complete.

Scientists spend a significant amount of time waiting for programs to run to completion. The distribution of waiting times is shown in Figure 2.2(a). Nearly half of the researchers who participated in the field study spend more than a few days waiting for program completion. Of this, around 15% of researchers wait on the order of months. While the researchers who ran programs for days and weeks were randomly distributed across different scientific disciplines, the researchers who waited for months were from three distinct departments: Chemistry, Geosciences, and Chemical Engineering. The corresponding programs in Chemistry and Chemical Engineering involved various forms of molecular dynamics simulations, although in each case, different scientific theories were leveraged. The program from Geosciences performed ocean modeling.

Around a third of the researchers wait on the order of a few hours for program completion while one-fifth waited on the order of minutes. Almost all the people who waited only for a few minutes used numerical computing environments like MATLAB to run their programs. A vast majority of these researchers were primarily experimentalists/theorists and employed computation to either validate theories or do data analysis. In the latter case, the analysis programs dealt with small data sets and were written by the researchers themselves. On the other hand, researchers who waited for days used an even mix of programs written in numerical computing languages and programs written in general purpose programming languages like C, C++ and FORTRAN and performed analysis over larger data sets along with other computational tasks like simulation and optimization.

Currently, the research conducted by 85% of researchers would profoundly change with faster computation. Nothing evoked stronger reactions during the interviews than questions regarding the impact of faster computation. Responses included "more papers, more quickly," "I'll graduate in 3 years instead of 5," "can get you out of school earlier," and "if it is 2x faster, life will be five times better." Several patterns regarding the potential areas of research improvement emerged during the field study, which can be concretely categorized as follows.

- **Accurate scientific modeling.** Across several disciplines, researchers approximate scientific models to reduce running-times. Faster computation enables accurate scientific modeling within time scales that are previously thought as unattainable. Across disciplines, an order of magnitude performance improvement was cited as a requirement for significant changes in research quality. For instance, researchers simulating precipitation/evaporation of earth science processes (Civil and Environmental Engineering), can adopt the use of finer resolution models which are currently avoided due to prohibitive communication costs between fine-grained cells and consequent increased time for convergence. A similar pattern holds for molecular dynamics simulations (Chemical Engineering), where increasing computation speed would not require researchers to relax error thresholds/step sizes to allow for faster convergence. The field study data reveals that 34% of the interviewees would directly benefit from accurate models.

- **Speeding up the scientific feedback loop.** Scientific computation is typically not performed in isolation, but as part of a three step feedback loop: (a) Evolve scientific models (b) Perform computation using models (c) Revise models based on computational results. For 30% of researchers, slower computation in Step (b) leads to an overall slow feedback loop. An example instance observed was in Computational

Biology, where different machine learning techniques are iterated over genome/protein data to predict gene interactions/protein structure. Faster computation shortens this feedback loop which in turn results in faster availability of prediction data to the larger scientific community.

- **Wider exploration of parameter space.** Currently, many researchers fit their scientific models to only a subset of available parameters for faster program runs. For instance, psychologists studying human decision making build models that fit only a sparse subset of parameters, despite the potential of obtaining accurate information about human subject by choosing a larger set of parameters. Other cases included the use of a faster but more approximate heuristic for determining shortest path problems in a stochastic network (Operations Research and Financial Engineering). Around 23% of researchers fall into this category. For these researchers, faster computation translates into better heuristics and eventually broadening their research scope to be more general and realistic.

- **Others.** For 12% of researchers, faster computation leads to better sensitivity analysis to data (lower normalized errors with repeated runs in parallel), faster post-processing of a huge amount of experimental data, and reduced effects of queuing time on simulation runs.

Over the past few decades, generations of newer parallel hardware have met the need of faster computation for scientific applications [85]. The continuing doubling of transistors as per Moore's Law [149], even in the presence of power and thermal walls for uniprocessor design, has meant that parallel hardware is now ubiquitous. Faster hardware will no doubt enable accurate scientific modeling and wider parameter space exploration for free in many applications. However, for an increasing number of applications, it is software and not hardware that is on the critical path of performance [214].

Despite enormous wait times, many scientists run their programs only on desktops.

Traditionally, large scale scientific computing problems have been solved by relying on powerful supercomputers, massively parallel computers, and compute clusters [115]. In practice, do scientists take advantage of these powerful computing resources when they are available and easily accessible?

While researchers use a wide variety of computational resources, a substantial portion (40%) of them only use desktops for their computation. This is despite the fact that the computational tasks performed take more than a few hours for completion, for more than half of these researchers. Although more than three-fourths of these desktops have multiple cores (most commonly two), almost all of them run only single threaded code. Furthermore, surveying the algorithms and computational techniques employed in these programs reveals that several of these are amenable to parallelization.

Around half of the interviewees use clusters, with about one-fifth leveraging both desktops and clusters in their research. Use of multiple clusters with varying architecture types was common in this group as well. Typically the same code was run unaltered on multiple machines. Only a small proportion of researchers used GPUs and shared-memory compute servers in their research. Servers offer a unified memory address space as opposed to a divided address space in the case of clusters. The servers were further characterized by the lack of a job submission system, whereas clusters usually had a job submission system for batch processing. The distribution of computational resources employed is shown in Figures 2.2(b) and 2.4.

A vast majority of subjects continually trade-off speed for accuracy of computation. However, even while doing so, scientists very rarely tune their applications to perform optimally on each specific cluster that they use for running them. Frequently seen patterns include increasing the error thresholds to allow for faster convergence of optimization

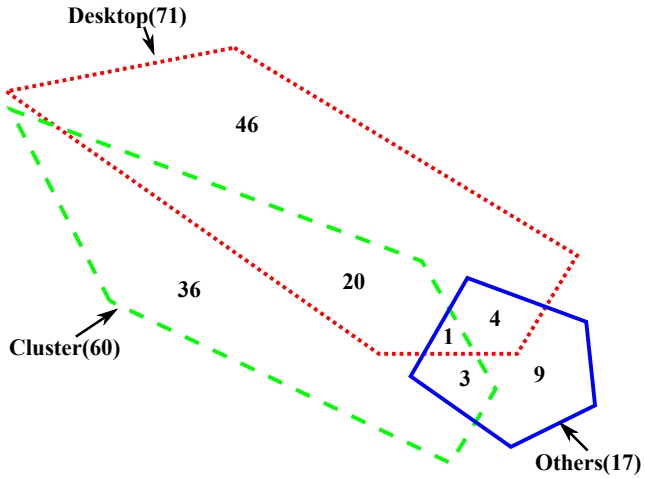


Figure 2.4: Proportional distribution of Computational Resource Usage. *Others* include Servers and GPUs

solvers, performing fewer simulation runs which made the results more susceptible to outliers, using coarse-grained scientific models that are greater approximations of the physical reality, and analyzing fewer data points to do hypothesis testing. Most of these choices were guided by the execution time witnessed on the first machine on which these programs were run. Some researchers while testing their initial implementations on their desktops wanted the runs to complete overnight. Often, these programs were run unaltered after an initial implementation on multiple machines with widely varying architectures.

2.2.4 Performance Enhancing Methods

Representative questions concerning this theme included: Do researchers care enough about performance to optimize their code? What tools and techniques are used to optimize for performance? Do researchers target the most time consuming portions of their code? Are researchers aware of parallelism and do they make use of different parallelization paradigms to deliver performance in their code?

Scientists do not optimize for the common case.

Compiler writers and computer architects are well acquainted with the adage "90% of the execution time is spent in 10% of the code." Therefore, the general belief is by optimizing hot code, greater performance gains can be expected per unit of effort. The results from the field study indicate that scientists have hypotheses on which portions of their code are hot, but more often than not do not test these hypotheses. Consequently, scientists may not be targeting the important sections of their program. A little more than half of the interviewees manually optimized code for performance. However, only 18% of researchers who optimize code leveraged profiling tools to inform their optimization plans. More than one-third of researchers were not aware of any profiling tools and did not time different portions of their code while nearly one-fourth of researchers had heard about profiling but did not take advantage of them.

The most common reasons given for not using profilers were "not a first order concern," "not heard of them," "will not help," and "I know where time is spent." As a result, quite a few scientists misjudged the portions of code that were the most time consuming and ended up optimizing code that did not contribute much to overall runtime. For example, one researcher performing biophysics simulations mistook MPI communication in his code for a performance bottleneck, even though it contributed to only 10% of the execution time. Profiling the application revealed that the real bottleneck was in a sorting algorithm, which accounted for more than 40% of execution time. Replacing the sorting algorithm with a faster one gave greater improvements in performance than optimizing for communication. Of the people who leveraged profiling information, close to three-fourths used a standard profiling tool like `gprof` and `MATLAB profiler`, while the remaining wrote custom timers (for example, function call counts rather than time spent in certain functions) in specific portions of code to profile for execution time.

Researchers employed a wide variety of performance enhancing strategies in their code as shown in Figure 2.2(c). These strategies can be grouped into two categories. High level optimizations (done by 28% of researchers) included algorithmic changes, choice of better data structures, and use of specialized libraries. Typical examples for these include better heuristics for optimization problems, choice of a concurrent data structure over sequential one, use of specialized digital signal processing engines and built-in libraries in MATLAB. Low-level optimizations (done by 26% researchers) included manual loop optimizations and use of compilation flags. The most popular of loop optimizations was manual loop vectorization. Of the people doing high-level optimizations, only around half actually profiled their code. Of the people doing loop optimizations, only a third ever profiled their code. A little less than 10% of researchers use compiler optimization flags (e.g., -O3). Many researchers were either unaware of compiler optimization flags or thought of compiler optimizations as too low-level to actually make a significant difference in performance.

Scientists are unaware of parallelization paradigms.

Very few researchers were aware of or took complete advantage of various parallelization paradigms like message passing and shared memory parallelism directly in their code. Less than one-third of researchers had heard about different forms of parallelism (data, task/functional, and pipeline) that could potentially be applicable to improve the performance of their code. About a third of researchers did not use any form of parallelism in their research at all. Half the researchers relied heavily on batch processing. Researchers who leveraged parallelization paradigms in their programs can be categorized as follows (Figure 2.2(d)):

- **Message Passing based Parallelization.**

Around 22% of researchers exploited message-passing-based parallelism in their research. Except for one researcher who used UNIX sockets directly, the rest of the researchers used MPI [86]. The vast majority of these people did not actually write MPI code themselves, but rather ran open source programs based on MPI. Many of them modified only the sequential portions of these programs and lacked knowledge about MPI concepts. Many researchers wanted to learn more about MPI via peer training and academic courses. Some of them had attended various two-day mini-courses offered by PicSCiE on MPI. However, they still had many complaints like "it is hard," "looks complex," "big learning curve," and "implementation time is too high."

Of the few researchers who wrote MPI code, most faced enormous problems debugging. Their experience using debuggers like gdb and TotalView was not good. All of them uniformly complained about the complexity of understanding and using such "unintuitive" tools on clusters. Researchers had significant difficulty exploiting parallel I/O on clusters. Quite often, they resorted to using a single process to do all the prints, even when it increased communication costs.

- **Thread based CPU Parallelization.** Despite the fact that most desktops today have at least two cores and almost all nodes in a cluster have multiple processors operating on a shared memory address space, only 7% of researchers leveraged any form of thread based shared memory CPU parallelism. A total of only eight researchers (of 114) admitted any knowledge of threads or had any experience using explicit thread based parallelism in their research.

- **GPU based Parallelism.** Around 9% of researchers leveraged GPUs. Of these, there was one expert per research group who wrote the GPU based parallel code and the rest of the researchers were users of this code. For instance, research staff studying

collective animal behavior in ecology and evolutionary biology included postdoctoral researchers with relevant computer science background who wrote the initial version of the code. Other researchers used GPU based open source versions of well known programs (for example, hoomd [18]).

- **Loop Based Parallelism.** Only 11% of researchers utilized loop-based parallelism, where programmer written annotations enable a parallelizing system to execute a loop in parallel. The most common form of loop based parallelism was the use of `parfor` construct in MATLAB, which enables execution of multiple iterations of a loop in parallel and requires that the iterations access/update disjoint memory locations.

Only one researcher exploited pipeline parallelism, even though pipeline parallelism is a natural way of extracting performance for many "simulate, then analyze" kind of applications. In these applications, a simulation run generates a lot of data (typically on the order of gigabytes), which is then analyzed by another piece of code after the simulation ends. Instead of sequentializing the simulation and analysis codes, creating a pipeline between these codes can achieve a significant overlap between them. This in turn would reduce the feedback cycle time drastically.

Even though researchers understand the importance of parallelism in accelerating their research, the predominant perception of parallel programming is that of black art ("heard it is notoriously hard to write" on MPI, "scared of it" on shared memory parallelism). The emerging heterogeneity in parallel architectures and explicit programming models is not helping either. Even though researchers seem excited at the potential of newer parallel architectures, they seem overwhelmed by current programming techniques for these machines ("GPUs can give 25x instantly instead of waiting for generations of Moore's Law, but I don't know how to program them," said a prominent scientist).

2.3 Summary

The results from the field study, as described in this chapter, make a strong case for the design, implementation and evolution of parallel programming solutions that require minimal programming effort for delivering performance. Although the importance of parallel performance cannot be overstated given the enormous running times of many scientific applications, one of the central results of the field study is that programming time is on the critical path of delivering on this performance. Specifically, solutions that preserve the sequential programming model as the default would be preferable since it is not likely to result in a steep learning curve for most scientific programmers.

Designing solutions for general purpose, imperative languages like C and C++ has the potential for making the maximum impact as evidenced by its widespread use across different scientific domains and more so in computationally intensive fields. Given the prevalent use of desktops for computational runs and the trend of increasing core counts on chip multiprocessors on these desktops, parallelization solutions that are no longer just focused on clusters but are agnostic to underlying machine architectures are preferable. These solutions preserve abstraction boundaries between software and hardware and preserve performance portability across successive generations of parallel architectures. The implicitly parallel programming (IPP) approach adopted in this dissertation has all the characteristics outlined above.

Chapter 3

Implicit Parallel Programming

This chapter describes and compares various state-of-the-art parallel programming and parallelization solutions. Table 3.1 gives an overview of the comparison. In this chapter, we use the term "automatic parallelization" to refer to compiler, runtime and other techniques for extracting parallelism from sequential code without requiring *any* intervention from a programmer.

We use the term "implicit parallel programming" to refer to a model of programming where parallelism is exposed implicitly as a result of certain design choices made at the language level, either in the form of specific high-level programming abstractions, extensions, or restrictions to sequential languages with no explicit support for manual parallelization. Therefore, the key distinction from the other approaches described in this chapter is that IPP does not require any special manual effort in low level parallel programming and the task of *how* to select the optimal *parallelization strategy* and *synchronization mechanism* is left to an automated tool-chain that includes a compiler and a runtime system.

We use the term "explicitly parallel" to refer to an approach that requires any of the following:

- a programmer to specify a parallelization strategy. For instance, the programming model may require a programmer to manually write code for specifying creation, management, and termination of a particular form of parallelism like task, data or pipeline parallelism.

- a programmer to specify a synchronization mechanism. For instance, the programming model may require a programmer to write code to manually specify specific forms of concurrency control mechanisms like transactions, locks, or condition variables.

- a programmer to explicitly specify thread partitioning. For instance, the programming model may require a programmer to write code to manually specify the code that constitutes the stages of a pipeline, and explicitly manage the flow of data between the stages by means of a producer-consumer library.

Finally, "interactive and assisted parallelization" refers to a methodology for reducing the effort of manual parallel programming by providing a programmer-friendly parallelization tool. The rest of this chapter describes each of these solutions in greater detail and makes a broad comparison between them and the IPP techniques proposed in this dissertation.

3.1 Automatic Parallelization Techniques

Automatic parallelization techniques based on compiler transformations can be classified into three main categories: (a) independent multithreading (IMT) (b) pipeline multithreading (PMT) and (c) cyclic multithreading (CMT). All these techniques extract loop level parallelism. The baseline versions of these techniques are non-speculative. When combined with speculation, these techniques have increased applicability and performance.

Methodology	Representative Systems	Default Prog. Model	Parallelization Strategy	Concurrency Control	Code Partitioning	Parallelization Assistance	Annotation Support
Automatic Parallelization	DOALL [17], Polaris [36] SUIF [191], DOACROSS [59] DOPIPE [62], DSWP [158] PS-DSWP [172] LOCALWRITE [92] Inspector-Executor [185] SpecDOALL [220] STMLite [143] LRPD [177], Spice [174] TLS [192], SpecDSWP [204]	Sequential	Auto	Auto	Auto	×	×
Implicit Parallel Programming	Physis [142], Chorus [139] Liszt [67], Paralax [206] VELOCITY [41], N-way [54] ALTER [203], pH [154] SQL [**?**], COMMSET, WEAKC	Sequential	Auto	Auto	Auto	✓	✓
Explicit Parallel Programming	Cilk [81], Cyclone [87] X10 [52], DPJ [37] Intel TBB [162], TPL [131] C++0x [29], Erlang [21] Atomos [47], AME [105] OpenTM [26], BOP [69] pthreads [43], MPI [86]	Parallel	Manual	Manual	Manual	×	×
Interactive and Assisted Parallelization	Parascope [119] SUIF Explorer [135] Tulipse [215] Intel Parallel Advisor [12] Dragon [99], Parameter [126] D Editor [100], PAT [188]	Parallel	Manual	Manual	Manual	✓	×

Table 3.1: Comparison of various parallelization paradigms. The programming effort is represented by the combination of (a) Default programming model (Parallel takes more effort than Sequential) and the entity responsible for specifying (b) the Parallelization Strategy (data, task, pipeline) (c) Concurrency Control (transactions, locks) (c) Code Partitioning (pipeline stages). In all cases, 'Manual' implies more effort than 'Auto'. Parallelization Assistance refers to tool support for determining impediments for parallelization and 'Annotation Support' refers to language support for removing artificial constraints on parallelizing programs.

Within this general categorization, there are several differences in the various state-of-the-art implementations depending on the exact mechanisms employed to either improve applicability or performance or both. The rest of this section describes many of these important automatic parallelization techniques and specifies how our IPP techniques relate to them.

IMT refers to extraction of parallelism from a loop with no loop-carried dependence. In the absence of loop carried dependences, a compiler can execute all iterations of a loop entirely in parallel without any need for synchronization. The DOALL [17] technique is

most popular of all independent multithreaded techniques. It was initially proposed for loops with regular memory accesses in FORTRAN based scientific computation. Early automatic parallelization systems like Polaris [36] and SUIF [191] supported DOALL. This technique has the highest performance potential due to the absence of any synchronization but has limited applicability due to its strong requirement of not having any inter-iteration dependences.

CMT refers to extraction of a particular form of parallelism from loops with loop-carried dependences where each iteration of the loop executes as a complete unit on a single thread or core while synchronizing any data dependence that exists from earlier iterations (threads) to later iterations (threads) to preserve sequential execution semantics. The DOACROSS [59] technique falls under this category and is supported within SUIF [191]. Cyclic multithreading has the advantage that it applies to more loops than DOALL but the synchronization overheads associated with this technique often result in poorer performance.

PMT refers to extraction of parallelism from loops that may contain loop-carried dependences into a pipeline of threads. In this technique, an iteration of a loop is split into different stages of a pipeline and each stage is scheduled to be run on a separate core/thread in such a way that all loop carried dependences are kept local within a thread and the intra-iteration dependences are communicated in a single direction from earlier stage to a later stage of the pipeline. Early proposals of DOPIPE [62] are an example of PMT.

The DSWP family of transformations [158, 172] which extract pipelined parallelism from loops with irregular memory accesses are a recently proposed set of PMT techniques. The primary insight behind the DSWP algorithms is that by keeping dependence cycles local, parallel computation is decoupled from communication and the parallelization is more latency tolerant. The original DSWP [158] algorithm extracted parallelism onto sequential stages, ie, each stage could be scheduled to run only on a single thread.

The PS-DSWP [172] technique extends DSWP to obtain greater scalability. It allows stages with no loop-carried dependences internal to them to run in parallel on multiple threads. Overall, the DSWP family of transformations are more applicable than DOALL and has much better performance characteristics than DOACROSS.

In addition to the above, other techniques such as LOCALWRITE [92], Inspector-Executor [185] and recent techniques based on matrix multiplication [183] have been proposed for specialized non-speculative parallelization of loops with irregular reductions. All the techniques described so far in this section rely on static analysis to determine the absence of dependencies in order to obtain a parallel schedule. Static analysis is inherently conservative due to its requirement to be sound across all inputs to a program. However, very often various kinds of dependences rarely manifest at runtime (for eg, error conditions) even though when viewed statically they can inhibit parallelization.

Various solutions have been proposed to enhance each of the above automatic parallelization techniques with speculation. The idea behind speculation is that by predicting that certain infrequent dependences (identified via profiling) do not manifest during execution, and by providing a runtime system that validates this prediction and recovers in case of a misprediction, sound and scalable parallelization of many programs with irregular memory access and control flow can be obtained. Most of the modern speculative parallelization systems depend on offline profiling [28, 217, 201] to determine dependences that manifest infrequently at runtime.

Speculative DOALL [220] is a system to extract speculative IMT parallelism and demonstrate its benefits for SPECfp programs. STMLite [143] is a speculative system for extracting DOALL parallelism that is based on a customized version of software transactional memory. The LRPD test [177] determines the criterion speculative parallelization of loops with iteration-local arrays and reductions. Privateer [109] is a recently proposed system

for speculative privatization and reductions that enable scalable speculative DOALL parallelization. Spice [174] is another technique that leverages value speculation for extracting DOALL parallelism.

Thread-level speculation (TLS) techniques present a speculative version of DOACROSS. While the initial versions of TLS [192, 193] required custom changes to hardware, recent proposals [155] implement TLS in software. Speculative Decoupled Software Pipelining (SpecDSWP) [204] is a technique that enhances DSWP with intelligent speculation where speculation is applied only to break longest few dependence cycles to increase both applicability and scalability of pipeline multithreading.

While automatic parallelization techniques require little or no effort from a programming perspective for extracting parallelism, they are constrained by the need to respect sequential semantics. Due to this, these techniques fail to achieve scalable speedups in many cases where the non-determinism present in a high-level algorithm is obscured by the artifacts of the sequential programming model.

3.2 Implicit Parallel Programming Approaches

Implicit parallel programming (IPP) addresses the problems associated with automatic parallelization described above. In particular, the IPP techniques proposed in this dissertation have automatic parallelization at its core. Given a sequential program, our IPP techniques first attempt to parallelize a program without any help from a programmer using some of the automatic parallelization techniques discussed above. In particular, our current implementation applies non-speculative versions of DOALL, DSWP, and PS-DSWP, and a specific form of OR parallelization. Our IPP annotations are applied only when these techniques fail to achieve scalable speedup. These IPP annotations exploit programmer knowledge about domain algorithms to relax the sequential semantics, and provide these

automatic parallelization techniques with additional freedom to scalably parallelize these relaxed programs.

This section reviews broadly related prior work in the area of implicit parallel programming, and places the IPP techniques proposed in this dissertation in context. An in-depth comparison of work related to the specific techniques proposed in this dissertation is made in Chapter 7.

A lot of work has been done on exploiting parallelism within functional programming languages that impose programming constraints at the language level [112, 154, 95, 61]. For instance, properties such as function purity, data structure immutability are enforced by default in many functional languages. In addition, for many of these languages the parallel and sequential semantics coincide. Feedback directed implicit parallelism systems [95] present a model for implicitly exploiting speculative parallelism within the runtime system of a lazily evaluated functional language. This approach employs profiling at runtime for feedback-directed extraction of parallelism transparently at runtime. Skeleton functions [61] serve as building blocks of a parallel program and provide a set of program transformations between these skeletons aimed at obtaining performance portable code.

Specific parallel programming frameworks have been proposed for effective parallelization of scientific code within certain domains. For instance, Physis [142] is a programming framework for translating high-level specifications for grid computation into a scalable parallel implementation in CUDA or MPI via compiler based techniques. Chorus [139] presents a high-level parallel programming approach based on "object assemblies" for mesh refinement and epidemic simulations. Liszt [67] is a domain specific language for building highly parallel mesh-based PDE solvers on a wide variety of heterogeneous architectures.

Two salient features distinguish the IPP techniques described in this dissertation from conventional IPP approaches described so far. Our extensions have the sequential programming model as the default without additional restrictions. This helps programmers since it is much easier to think sequentially rather than in parallel. In addition, eliding the IPP extensions results in a deterministic, sequential output and the IPP extensions provide a way to incrementally relax sequential semantics and implicitly introduce parallelism. Secondly, our IPP extensions are semantic in nature, and their use changes the output of the program. They require programmers intervention as no static analysis tool can deduce these extensions automatically. Thus, these extensions are not redundant, and by involving the programmer only when necessary in breaking artificial dependences that restrict automatic parallelization, our IPP methodology significantly reduces the burden on the programmer.

Similar to the language extensions for IPP proposed in this dissertation, similar proposals for semantic language extensions have been made in the past. Jade [179], Galois [128], DPJ [38], Paralax [206], and VELOCITY [41] propose various forms of semantic commutativity annotations with various degrees of expressiveness. Chapters 4 and 7 make an in-depth comparison of these proposals with COMMSET, the semantic commutativity based construct and IPP system that is proposed in this dissertation. Similarly, apart from the above-mentioned semantic commutativity proposals, N-way [54], Cilk++ hyperobjects [79], and ALTER [203] present programming extensions and systems that could be applied for weakening strong consistency requirements on sequential data structures. Chapters 5 and 7 make an in-depth comparison of these proposals with weak consistency data structures proposed in this dissertation.

3.3 Explicitly Parallel Approaches

Despite the advances made in the field of automatic parallelization and implicit parallel programming, the prevalent paradigm for parallelization today is still explicitly parallel. In the last few decades, many explicitly parallel languages have been invented that give a programmer a lot of freedom with respect to parallelization. However, this comes at the cost of increased programming effort. This section describes recent and past approaches to explicit parallel programming.

Cilk [81] is an explicitly parallel, multithreaded extension of C that provides primitives for parallel control. In particular, it relies on a programmer to explicitly specify which functions can be executed in parallel and when threads can be joined using *spawn* and *sync* keywords. Cilk also provides an analytical model for performance based on the concepts of "work" and "critical path length". Similar to the IPP techniques presented in this dissertation, Cilk has the nice property that eliding the extensions results in a sequential program with well defined serial semantics. Cyclone [87] is another extension of C with multithreading, which additionally also guarantees safety from race conditions if a program type checks. Kawaguchi [116] presents a similar system of dependent type system that guarantees deterministic parallelism. Our IPP techniques relax strict determinism requirements imposed by a sequential programming model to accurately reflect application semantics.

X10 [52] provides an object oriented (OO) approach to parallelism at the language level and has a memory model that is partitioned within a global address space. It provides OO primitives like *foreach*, *ateach*, *future*, and *async* that a programmer can use to specify the units of work to be executed in parallel and also specify how the work has to be partitioned to execute on separate memory regions within a global address space. Our IPP techniques rely on compiler and runtime support for partitioning both memory and tasks for parallel execution.

DPJ [37] is another language that presents a region-based memory model for the purposes of safe deterministic parallelism (see Chapter 7 for a complete comparison with COMMSET). Ribbons [101] is a recently proposed programming model that supports full or partial heap isolation for shared memory parallel programming.

Futures [213] represent a piece of computation that can be computed asynchronously or lazily in parallel with the main computation and the result used when needed within the main program. Welc et al. [213] present a programming model and an API for safe futures for Java. In this model, a programmer needs to explicitly choose annotate functions that can execute in parallel with a program's main computation as futures, and a runtime system guarantees preservation of sequential semantics in its execution. Our IPP techniques relax sequential semantics, and our annotations declare a semantic property and are not imperative injunctions for creating parallel tasks.

Intel Threaded Building Blocks(TBB) [162] is a C++ runtime library that provides object oriented abstractions, primarily using template based generic programming, to specify the creation and management of parallelism at a higher level than provided by the POSIX threading primitives. TBB provides abstractions for programmers to create both data and pipeline parallelism by wrapping sections of code within appropriate functors and instantiating templates that perform the functionality of *foreach* keyword in Cilk. Additionally, the runtime library provides functionality to adapt the number of threads and provide load balancing at runtime. The Task Parallel Library (TPL) [131] provides a similar framework to TBB but for a managed language. While our IPP compiler does not rely on a programmer for generating pipeline and data parallel schedules, the load balancing functionality of TBB and TPL could be beneficial in its backend.

The newest C++ standard provides support for concurrency [29] at the language level. In particular, the proposed concurrency primitives are based on threading using functors (`std::thread`) within the C++ standard library. Programmers can use these facilities

to explicitly create threads, manage the concurrency with condition variables, mutual exclusion primitives, and memory barriers, and terminate them with halt primitives provided within the standard.

Several parallel extensions of functional languages are explicitly parallel. Concurrent ML [178] is an extension of ML with support for concurrency via message passing. Erlang [21] is another language that is functional and parallel by default and also relies on message passing. In contrast, our IPP annotations are designed and implemented as extensions to imperative languages like C/C++ and can be easily ported over to functional languages.

The map-reduce paradigm [65, 175] is an application of a well-known functional programming idiom for large scale, fault tolerant parallel data processing over clusters of machines. It imposes a strict two-phase (DOALL and reduce) programming model and presents a shared memory abstraction to the programmer. Parallelization and load balancing are hidden from the programmer. FlumeJava [50] is a generalization of the MapReduce API that enables easier development of efficient data-parallel pipelines. NESL [34] is a data-parallel language that is functional at its core. Data Parallel Haskell [49] and pH [154] are extensions to Haskell that support nested data parallelism.

The Berkeley dwarfs [22] represent a pattern language for parallel programming. This language is based on 13 core patterns for parallel programming (13 "dwarfs") and reflect the design patterns based approach to software engineering. The language supports a wide variety of parallelization paradigms based on representative application use cases from a variety of computational domains. It also envisions a larger role for "autotuners" to maximize performance while minimizing programming effort. While this approach is similar to a library based approach, our IPP techniques have been integrated with autotuning frameworks, one as part of the Parcae [171] system for flexible parallel execution, and another within WEAKC (see Chapter 5).

Several solutions have been proposed that support transactional memory at the language level. Atomos [47] is a language that is derived from Java but replaces Java's synchronization constructs with transactional memory constructs. Although programmers no longer need to use low level synchronization primitives like locks and condition variables, the task of identifying and parallelizing profitable code sections still has to be manually performed. Transaction Collection Classes [46] provide a transactional replacement for Java collection classes. Automatic Mutual Exclusion [105] provides a programming model based on transactions where all state is implicitly shared between different threads by default unless explicitly specified using mutual exclusion primitives by a programmer. In our IPP system, transactional memory is used as one of synchronization mechanisms in the backend whenever applicable and is not exposed to the programmer.

OpenTM [26] extends the OpenMP programming interface with primitives for transaction based synchronization and parallelism. Even though programmers have to identify the individual units of work that need to be executed in parallel and specify the code sections where synchronization need be enforced, the OpenTM compiler and runtime system is free to ignore or vary the degree of parallelism at runtime. Transactional constructs for C/C++ [152] present a similiar model. Like OpenTM, our IPP annotations use `pragmas` and serve as hints that can be readily ignored. However, our approach does not require programmers to specify either a parallelization strategy or synchronization mechanism.

Safe programmble speculative systems [166] present programmatic constructs for safely exploiting value speculation to parallelize seemingly sequential applications. Sofware BOP [69] is a similar system that relies on value speculation but does not guarantee any particular ordering in its parallel execution. Recent proposals [117, 32] add support for ordering and determinism. Currently, our IPP techniques do not exploit value speculation, but they can be integrated transparently with prior compiler techniques [174].

OpenMP [51] is a popular programming model for shared memory parallel programming. It provides a set of compiler directives (*pragmas*) that can be applied on a sequential program and relies on associated compiler support and a runtime system to create and optimize parallelism during execution. Although OpenMP provides a relatively higher-level programming interface than compared to pthreads [43] or MPI [86], programmers still have to identify profitable sections of code, determine the best parallelization strategy and use specific synchronization mechanisms with differing semantics (using keywords like *atomic*, *critical*, *ordered*).

POSIX threads [43] provides a highly flexible, relatively low level support for explicit parallel programming for shared memory systems using a library based approach. The pthreads API is flexible which programmers can use to manually create any form of parallelism – task, data, or pipeline parallelism. It also provides a wide variety of low level synchronization constructs ranging from different kind of locking primitives (mutexes, semaphores) to condition variables and so on. Although the expressiveness obtained by composing many of these primitives is high, amount of programmer effort required is high and the parallel programming pitfalls like race conditions, deadlocks, atomicity violations are quite common when using this API.

MPI [86] is a message passing API that is targeted toward cluster-based systems containing nodes with disjoint memory spaces. Similar to *pthreads*, the API is extremely flexible and expressive, with various routines for point-to-point and broadcast communication. While it is relatively easy to exploit data parallelism with MPI, exploiting task and pipeline parallelism can be quite cumbersome since synchronization has to be explicitly implemented manually via message passing protocols.

Many languages have been proposed for effectively programming high-performance streaming applications. StreamIt [200] is one such language that provides a structured model for streams with filters, and various mechanisms to composing these filters such as

a pipeline, split-join, and feedback loops. The language is supported by a compiler that has many stream specific analyses and optimizations. StreamFlex [190] is a recently proposed streaming language for Java. Several techniques have been also been proposed at the language level [88, 210] to efficiently map stream parallelism on to multicore. Compared to these, our IPP techniques rely on the DSWP family of compiler transformations [158, 204, 172] which considerably lessen the burden on the programmer.

Several parallel programming frameworks have been proposed for heterogeneous architectures. Merge [137] provides a library-based approach for programming heterogeneous architectures using the map-reduce approach, and a compiler converts a map-reduce program into low-level parallel codes for GPUs and other architectures. Lime [23] is a language with many built-in safety features for synthesizing parallel programs on heterogeneous architectures and presents a task programming model with support for split-join and streams. Compared to these, our IPP techniques are at a relatively high-level, architecture-agnostic, and not restricted to a particular parallelization methodology.

3.4 Interactive and Assisted Parallelization

Implicit Parallel Programming takes complete advantage of automatic parallelization by providing annotation support to remove artificial constraints that cannot be automatically deduced. Interactive and Assisted Parallelization provides a methodology for reducing the effort of manual parallel programming by providing a programmer-friendly parallelization tool. This tool is usually interactive and provides an user interface that describes various program properties relevant for parallelization at the program source level or in general at a high level. The user interface is usually part of an integrated development environment and gets its information from various backend modules like a compiler, runtime system or an analysis tool. Some of the tasks that the interactive tool assists a programmer with are in

identifying hotspots for parallelization, identifying performance bottlenecks in explicitly parallelized programs, eliciting programmer's help in identifying spurious dependences, and in performing some hard-to-do automatic transformations manually. This section describes recent and past approaches to interactive and assisted parallelization.

Parascope [119] represents one of the earliest tools for interactive parallel programming. It provided an intelligent interactive editor to support parallel programming by a combination of program analysis tools and expert advice via interactive editing. It was originally aimed at parallel FORTRAN programs that have regular array based accesses. Incremental parallelization of a program was supported by allowing a programmer to delete spurious dependences, ignore dependence sets, filter certain dependences all of which were displayed visually. Additionally, the tool could be used to point out the potential hazards in programmer-written parallel code using a combination of analysis of program text and program structure. Once parallelized, the tool could be used to give feedback on the different parallelization strategies by providing profitability estimates.

The D Editor [100], similar to Parascope, is an interactive tool for aiding parallel programming using Fortran D and High Performance Fortran. In particular, the D Editor provided assistance with dependence filtering, and guidance for a programmer on program partitioning, parallelism finding, effective placement of data placement annotations, and optimization of MPI communications.

SUIF Explorer [135] is an interactive parallelizer that focuses on minimizing the number of lines of code on which it requires programmer assistance for parallelization. It does so by utilizing precise interprocedural analysis to eliminate spurious dependences of which array dependence analysis is useful in particular. Another important component is the technique of program slicing [212] that enables reduction in the amount of code a programmer has to peruse. The Explorer provides a "parallelization guru" that combines static and dynamic information to identify hotspots for parallelization and also identifies the optimal

granularity for parallelization. Whenever analysis is imprecise, SUIF explorer also allows a programmer to specify that specific dependences do not exist and also to manually write privatization-code when automatic code transformation is hard.

PAT [188] is an incremental parallelizing tool that acts on both sequential and partially parallel FORTRAN programs. It provides guidance to a programmer to safely add explicitly parallel directives to existing FORTRAN programs. The main conceptual novelty of PAT is the incremental nature of its dependence analysis, ie, its ability to incrementally update dependence information as a programmer changes source code for parallelization. This ability plays a crucial role in providing a seamless user experience for interactive parallelization.

Dependence visualization tools [186] similar to PAT have been implemented as a plugin to an integrated development environment, for on-the-fly dependence visualization in assisting programmers with parallelization. Tulipse [215] is another tool for dependence visualization, especially for dependences that span loop nests, also provided as a plugin to an integrated development environment. Intel Parallel Advisor [12] is a recently released tool that helps developers to simplify and speed parallel application design and implementation.

Balasundaram [27] presents an interactive parallelization tool specifically targeted towards numerical scientific programs. In addition to developing and applying a general theory of dependence testing for pipeline parallelization, it provides support for synchronization analysis. Also, it provides debugging support for identification of race conditions.

Parallel programming environments for OpenMP [160] provide tools for interactive program parallelization using OpenMP. These tools help a programmer with program structure visualization, performance modeling, and with general parallel tuning. The first step in this approach is to put the program through a parallelizing compiler. Only when it fails

to uncover sufficient parallelism, it resorts to parallelization with a programmer's assistance. Dragon [99] is another tool that provides dynamic information for feedback-driven OpenMP-based parallelization in addition to assistance through static analysis.

Dig [68] advocates a refactoring approach to parallelism where a programmer begins with a sequential program, incrementally and interactively parallelizes it by selecting sections of code to be refactored while relying on analysis tools to validate the thread-safety of each incremental step. In addition to improving programmer productivity and performance, this approach is also aimed at improving code portability. Dig also describes different forms of refactoring each aimed at optimizing throughput, scalability, or thread safety.

Krelim [82] is a tool that exclusively focuses on identifying profitable sections of code for parallelization using profiling. It uses an analysis called the hierarchical critical path analysis, and provides a parallelism planner for ranking the different code regions in order of their profitability. Kismet [107] extends this work with sequence regions that separate instruction-level parallelism from other forms of parallelism.

Hawkeye [202] is a dynamic dependence analysis tool that assists programmers in identifying impediments to parallelization. In contrast to other such tools, Hawkeye tracks dependences based on the abstract semantics of well-defined data types, and ignores dependences that manifest as implementation artifacts. Parameter [126] is a tool that produces parallelism profiles for irregular programs, depicting the amount of parallelism when implementation artifacts are overridden.

The IPP techniques proposed in this dissertation are similar in many respects to the interactive and assisted parallelization techniques described earlier. Similar to SUIF explorer and other tools, our IPP techniques also have at its backend a parallelizing compiler and aim at minimizing the number of lines of code of programmer assistance. Similar to dependence visualization tools, our IPP system provides a frontend that displays program

properties at the source level in HTML to programmers to determine beneficial points in the source for insertion of semantic annotations. However, there are key distinctions:

1. Unlike interactive tools, our IPP techniques provide a rich and general semantic annotation language that enables programmers to specify relaxations of sequential program behavior in expressive ways. In addition, our backend provides associated support for understanding these annotations to aid in transparent parallelization in the "write once, optimize many" model.

2. Most of the interactive parallelization tools described above require programmers to eventually perform some form of explicit parallelization. By contrast, the annotations provided by our IPP techniques are semantic in nature and enable programmers to succinctly express high-level program properties without worrying about the details of *how* parallelization is achieved.

Chapter 4

Generalized Semantic Commutativity

Several class of emerging scientific applications contain computations involving random number generators, hash tables, binary search trees, list data structures, file I/O involving sequences of function calls (e.g., `fread` and `fwrite` to different files) that can be executed out of order (or commute with each other). For instance, gene sequencing applications frequently read and write data files of several gigabytes of data. Monte Carlo algorithms employed in simulation programs in several fields of science of engineering frequently call various forms of random number generation routines. Clustering applications employed in computational biology employ tree data structures where different orders of insertion operations result in different internal structures, although semantically representing the same cluster. However, such applications cannot be automatically parallelized due to artificial constraints imposed by the sequential model – all the above computations involve flow dependences on the internal state that cannot be broken without changing the output of the program, even though alternate outputs may be legal according to the semantics of the application.

Recent work has shown the importance of programmer specified semantic commutativity assertions in exposing parallelism implicitly in code [38, 41, 128, 179, 205]. The

IPP System	Concept				
	Expressiveness of Commutativity Specification				Requires Additional Extensions
	Predication		Commuting Blocks	Group Commutativity	
	Interface	Client			
Jade [179]	×	×	×	×	Yes
Galois [128]	✓	×	×	×	Yes
DPJ [38]	×	×	×	×	Yes
Paralax [205]	×	×	×	×	No
VELOCITY [41]	×	×	×	×	No
COMMSET	✓	✓	✓	✓	No

Table 4.1: Comparison between the COMMSET concept and other parallel models based on semantic commutativity

IPP System	Specific Parallel Implementation					
	Parallelism Forms			Concurrency Control Mechanism	Parallelization Driver	Optimistic Parallelism
	Task	Pipelined	Data			
Jade [179]	✓	✓	×	Automatic	Runtime	×
Galois [128]	×	×	✓	Manual	Runtime	✓
DPJ [38]	✓	×	✓	Manual	Programmer	×
Paralax [205]	×	✓	×	Automatic	Compiler	×
VELOCITY [41]	×	✓	×	Automatic	Compiler	✓
COMMSET	×	✓	✓	Automatic	Compiler	×

Table 4.2: Comparison between the COMMSET implementation and other parallel models based on semantic commutativity

programmer relaxes the order of execution of certain functions that read and modify mutable state, by specifying that they legally *commute* with each other despite violating existing partial orders. Parallelization tools exploit this relaxation to extract performance by permitting behaviors prohibited under a sequential programming model. However, existing solutions based on semantic commutativity either have limited expressiveness or require programmers to use additional parallelism constructs.

4.1 Limitations of Prior Work

Tables 4.1 and 4.2 shows existing parallel programming models based on semantic commutativity and compares them with COMMSET, the solution proposed in this dissertation. Jade [179] supports object-level commuting assertions to specify commutativity between every pair of operations on an object. However, Jade requires programmer written read-/write specifications for exploiting task and pipeline parallelism, which can be quite tedious to write. Galois [128], a runtime system for optimistic parallelism, leverages commutativity assertions on method interfaces. However, it requires programmers to use special set abstractions with non-standard semantics to enable data parallelism. DPJ [38], an explicitly parallel extension of Java uses commutativity annotations to override restrictions placed by the type and effect system of Java. However, these annotations can only be made at method interfaces within library classes. Several researchers have also applied commutativity properties for semantic concurrency control in explicitly parallel settings [46, 125]. Paralax [205] and VELOCITY [40, 41] exploit self-commutativity at the interface level to enable pipelined parallelization. VELOCITY also provides special semantics for commutativity between pairs of memory allocation routines for use in speculative parallelization. However, these commutativity constructs can only be used to specify self-commutativity of functions at their interfaces and hence are limited in terms of their expressiveness. This dissertation proposes an implicit parallel programming model based on semantic commutativity, called Commutative Set (COMMSET), that generalizes all the above semantic commutativity constructs and enables multiple forms of parallelism from the same specification.

The main advantages of COMMSET over existing approaches are: (a) COMMSET's commutativity construct is more general than others. Prior approaches allow commutativity assertions only on interface declarations. However, commutativity can be a property

of client code as well as code behind a library interface. COMMSET allows the commutativity assertions between arbitrary structured code blocks in client code as well as on interfaces, much like the `synchronized` keyword in Java. It also allows commutativity to be predicated on variables in a client's program state, rather than just function arguments as in earlier approaches. (b) COMMSET specifications between a group of functions are syntactically succinct, having linear specification complexity rather than quadratic as required by existing approaches. (c) COMMSET presents an implicit parallel programming solution that enables both pipeline and data parallelism without requiring any additional parallelism constructs. Existing approaches use parallelism constructs that tightly couple parallelization strategy with concrete program semantics, in contravention of the principle of "separation of concerns." In contrast, using only COMMSET primitives in our model, parallelism can be implicitly specified at a semantic level and is independent of a specific form or concurrency control mechanism.

The subsequent sections of this chapter describe:

1. The design, syntax, and semantics of COMMSET, a novel programming extension that generalizes, in a syntactically succinct form, various existing notions of semantic commutativity.

2. An end-to-end implementation of COMMSET within a parallelizing compiler that includes the front-end, static analysis to enhance the program dependence graph with commutativity properties, passes to enable data and pipeline parallelizations, and automatic concurrency control.

A detailed experimental evaluation of the COMMSET programming model on a set of twenty programs and programs collected from the field study is given in Chapter 6.

4.2 Motivating Example

Figure 4.1 shows a code snippet from a sequential implementation of md5sum (plus highlighted `pragma` directives introduced for COMMSET that are discussed later). The main loop iterates through a set of input files, computing and printing a message digest for each file. Each iteration opens a file using a call to `fopen`, then calls the `mdfile` function which, in turn, reads the file's contents via calls to `fread` and then computes the digest. The main loop prints the digest to the console and closes the file by calling `fclose` on the file pointer. Although it is clear that digests of individual files can be safely computed out of order, a parallelizing tool cannot infer this automatically without knowing the client specific semantics of I/O calls due to its externally visible side effects. However, the loop can be parallelized if the commuting behaviors of `fopen`, `fread`, `fclose`, and `print_digest` on *distinct* files are conveyed to the parallelizing tool.

One way to specify commuting behavior is at the interface declarations of file operations. Galois [128] extracts optimistic parallelism by exploiting semantic commutativity assertions specified between pairs of library methods at their interface declarations. These assertions can optionally be predicated on their arguments. To indicate the commuting behaviors of calls on distinct files, one would ideally like to predicate commutativity on the filename. Since only `fopen` takes in the filename as an argument, this is not possible. Another approach is to predicate commutativity on the file pointer `fp` that is returned by `fopen`. Apart from the fact that expensive runtime checks are required to validate the assertions *before* executing the I/O calls (which are now on the critical path), this approach may prevent certain valid commuting orders due to recycling of file pointers. Operations on two distinct files at different points in time that happen to use the same file pointer value are now not allowed to commute. This solution is also not valid for all clients. Consider the

[1]The commutativity specifications languages of Galois, Paralax, VELOCITY and COMMSET are conceptually amenable to task and speculative parallelism

following sequence of calls by a client that writes to a file (`fp1`) and subsequently reads from it (`fp2`) in the next iteration: `fwrite(fp1)`, `fclose(fp1)`, `fopen(fp2)`, `fread(fp2)`. Here, even though `fp1` and `fp2` may have different runtime values, they still may be pointing to the same file. Commuting `fopen(fp2)`, `fread(fp2)` with `fclose(fp1)` may cause a read from the file before the write file stream has been completed. Approaches that annotate data (file pointer `fp` in this case) to implicitly assert commutativity between all pairs of operations on that file pointer [179], run into the same problem.

Allowing predication on the client's program state can solve the above problem for md5sum. Since the input files naturally map to different values of the induction variable, predicating commutativity on the induction variable (in the client) solves the problems associated with interface based predication. First, no legal commuting behavior is prohibited since induction variables are definitely different on each iteration. Second, runtime checks for commutativity assertions are avoided since the compiler can use static analysis to symbolically interpret predicates that are functions of the induction variable to prove commutativity on separate iterations.

In order to continue using commutativity specifications on function declarations while still predicating on variables in client state, programmers either have to change existing interfaces or create new wrapper functions to take in those variables as arguments. Changing the interface breaks modularity since other clients which do not want commutative semantics are now forced to pass in additional dummy arguments to prevent commuting behaviors. Creating wrapper functions involve additional programmer effort, especially while replicating functions along entire call paths. In the running example, the `mdfile` interface has to be changed to take in the induction variable as an argument, to allow for predicated commutativity of `fread` calls with other file operations (`fopen` and `fclose`) in the main loop.

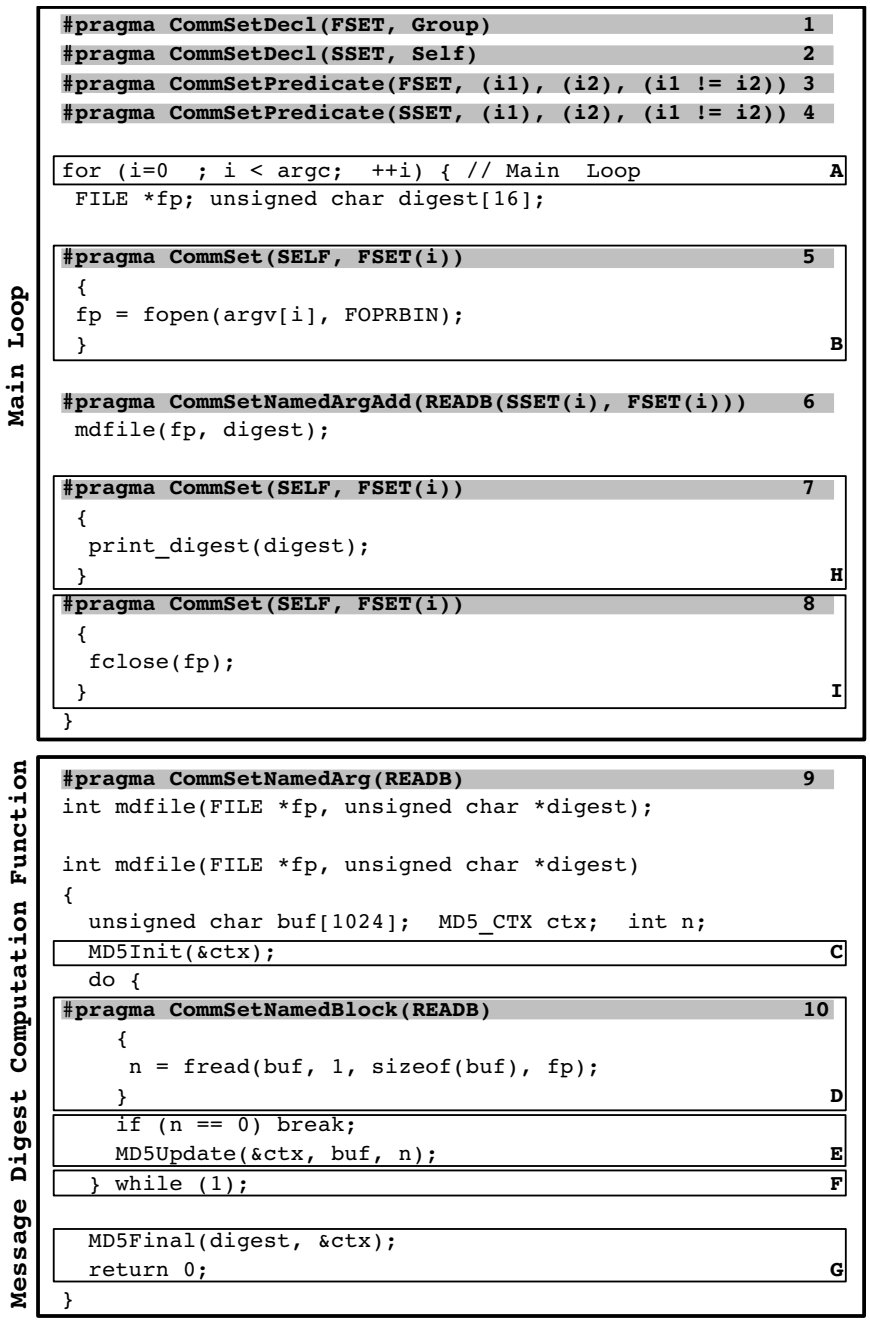


Figure 4.1: Sequential version of md5sum extended with COMMSET

The additional programmer effort in creating wrappers can be avoided by allowing structured code blocks enclosing the call sites to commute with each other. This is easily achieved in md5sum by enclosing the call sites of `fopen`, `fread`, `print_digest`,

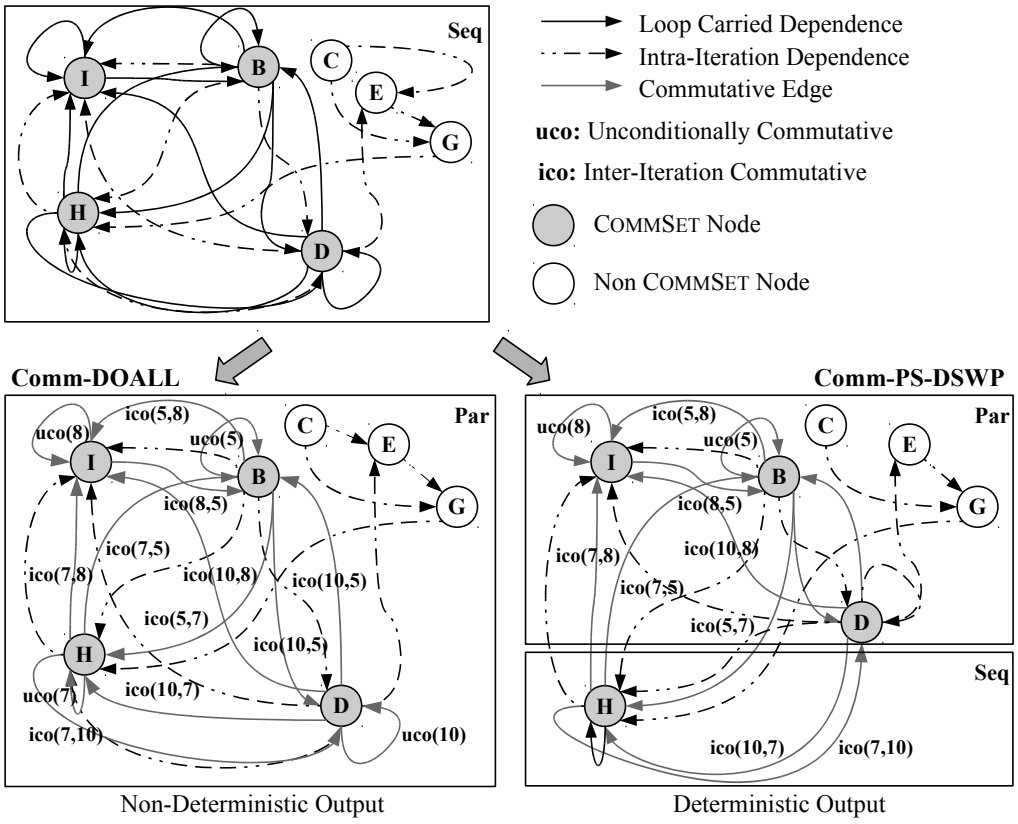


Figure 4.2: Program Dependence Graph for md5sum with COMMSET extensions

and `fclose` within anonymous commutative code blocks. Commutativity between multiple anonymous code blocks can be specified easily by adding the code blocks to a named set, at the beginning of their lexical scope. Grouping commutative code blocks or functions into a set, as presented here, has linear specification complexity. In contrast, existing approaches [38, 128] require specifying commutativity between pairs of functions individually leading to quadratic specification complexity. The modularity problem (mentioned above) can be solved by allowing optionally commuting code blocks and exporting the option at the interface, without changing interface arguments. Clients can then enable the commutativity of the code blocks if it is in line with their intended semantics, otherwise default sequential behavior is preserved. In the case of md5sum, the `fread` call can be made a part of an optionally commuting block. The option is exported at the interface

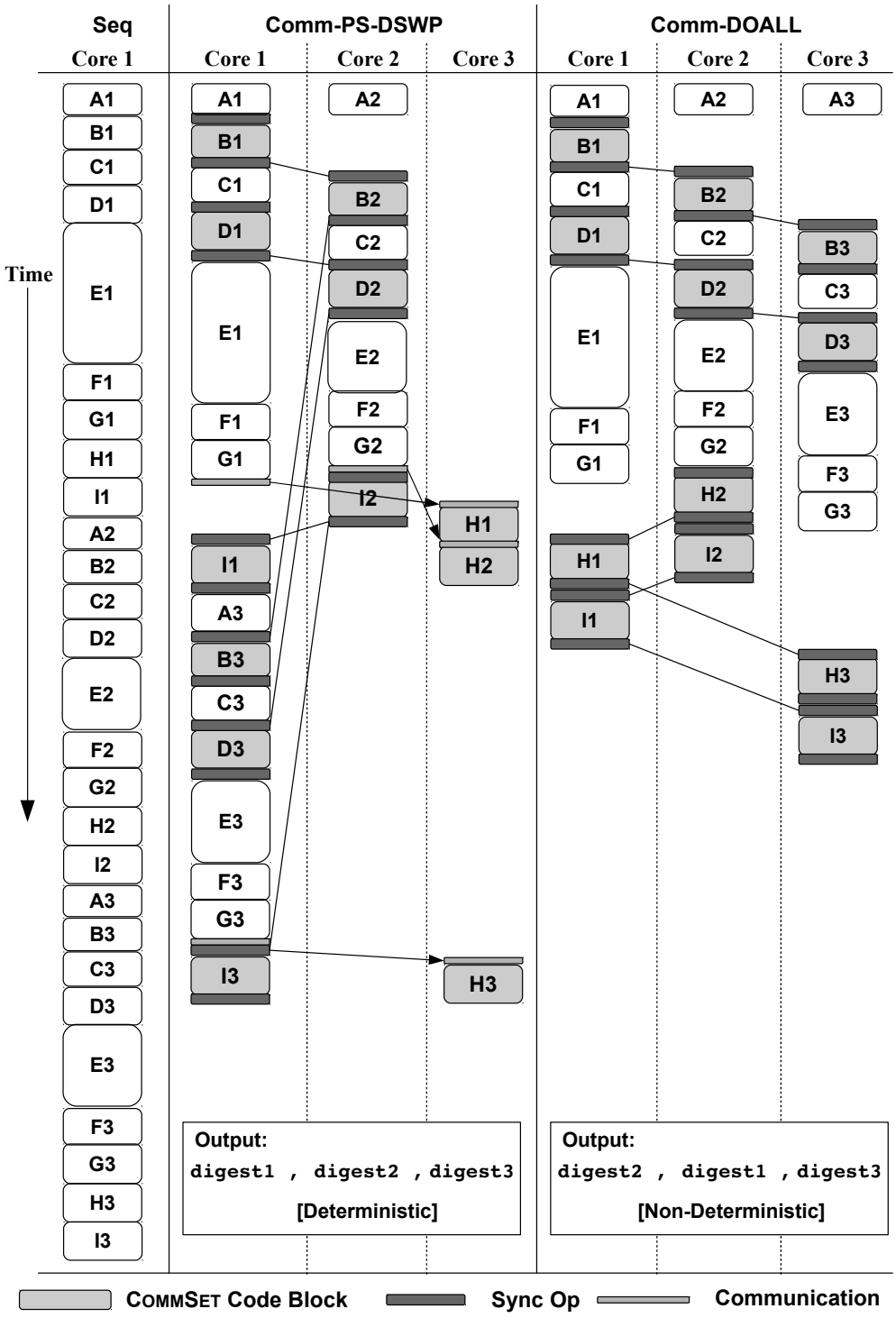


Figure 4.3: Timeline for md5sum Parallelizations

declaration of `mdfile` and is enabled by the main loop while other clients that require sequential semantics can ignore the option.

A commuting code block gives the programmer the flexibility to choose the extent to which partial orders can be relaxed in a program. This, in turn, determines the amount of freedom a parallelizing tool has, to extract parallelism. For instance, enclosing `fread` calls inside `mdfile` within a commuting block gives more freedom to a parallelizing tool to extract performance, as opposed to the outer call to `mdfile`. Depending on the intended semantics, the programmer can choose the right granularity for the commuting blocks which a parallelizing system should automatically guarantee to be atomic. Commuting blocks allow for a declarative specification of concurrency which a parallelizing tool can exploit to automatically select the concurrency mechanism that performs best for a given application. Automatic concurrency control has the advantage of not requiring invasive changes to application code when newer mechanisms become available on a particular hardware substrate.

Parallelizing tools should be able to leverage the partial orders specified via commutativity assertions without requiring the programmer to specify parallelization strategies. Existing runtime approaches require the use of additional programming extensions that couple parallelization strategies to program semantics. Galois [128] requires the use of set iterators that constrain parallelization to data parallelism. Jade [179] requires the use of task parallel constructs. DPJ [38] uses explicitly parallel constructs for task and data parallelism. COMMSET does not require such additional extensions. For instance, in md5sum, a programmer requiring deterministic output for the digests, should be able to express the intended semantics without being concerned about parallelization. The implementation should be able to automatically change to the best parallelization strategy given the new semantics. Returning to md5sum, specifying that `print_digest` commutes with the other I/O operations, but not with itself constrains output to be deterministic. Given the

new semantics, the compiler automatically switches from a better performing data parallel execution of the loop to a slightly less performing (in this case) pipelined execution. In the data parallel execution, each iteration of the loop executes in parallel with other iterations. In a pipeline execution, an iteration is split into stages, with the message digests computed in parallel in earlier stages of the pipeline being communicated to a sequential stage that prints the digest in order to the console.

Parallelization within the compiler is based on the Program Dependence Graph (PDG) structure [78]. Figure 4.2 shows the simplified PDG for sequential md5sum. Each labeled code block is represented by a node in the PDG and a directed edge from a node n_1 to n_2 indicates that n_2 is dependent on n_1. Parallelizing transforms (e.g. DOALL [118] and PS-DSWP [173]) partition the PDG and schedule nodes onto different threads, with dependences spanning threads automatically respected by insertion of communication and/or synchronization operations. With the original PDG, DOALL and PS-DSWP cannot be directly applied due to a cycle with loop carried dependences: $B \rightarrow D \rightarrow H \rightarrow I \rightarrow B$ and self loops around each node in the cycle. The COMMSET extensions help the compiler to relax these parallelism-inhibiting dependences, thereby enabling wider application of existing parallelizing transforms.

Figure 4.3 shows three schedules with different performance characteristics for md5sum execution. The first corresponds to sequential execution and the other two parallel schedules are enabled by COMMSET. Each of these schedules corresponds to three different semantics specified by the programmer. The sequential execution corresponds to the in-order execution of all I/O operations, as implied by the unannotated program. The PS-DSWP schedule corresponds to parallel computation of message digests overlapped with the sequential in-order execution of `print_digest` calls. Finally, the DOALL schedule corresponds to out-of-order execution of digest computation as well `print_digests`. Every COMMSET block in both the DOALL and PS-DSWP schedules is synchronized by the use

of locks (provided by libc) while PS-DSWP has additional communication operations. The DOALL schedule achieves a speedup of 7.6x on eight threads over sequential execution while PS-DSWP schedule gives a speedup of 5.8x. The PS-DSWP schedule is the result of one less COMMSET annotation than the DOALL schedule. The timeline in Figure 4.3 illustrates the impact of the semantic choices a programmer makes on the freedom provided to a parallelizing tool to enable well performing parallelizations. In essence, the COMMSET model allows a programmer to concentrate on the high-level program semantics while leaving the task of determining the best parallelization strategy and synchronization mechanism to the compiler. In doing so, it opens up a parallel performance optimization space which can be systematically explored by automatic tools.

4.3 Semantics

This section describes the semantics of various COMMSET features.

Self and Group Commutative Sets. The simplest form of semantic commutativity is a function commuting with itself. A *Self* COMMSET is defined as a singleton set with a code block that is self-commutative. An instantiation of this COMMSET allows for reordering dynamic invocation sequences of the code block. A straightforward extension of self-commutativity that allows commutativity between pairs of functions has quadratic specification complexity. Grouping a set of commuting functions under a name can reduce the specification burden. However, it needs to account for the case when a function commutes with other functions, but not with itself. For instance, the `insert()` method in a STL `vector` implementation does not commute with itself but commutes with `search()` on different arguments. A *Group* COMMSET is a set of code blocks where pairs of blocks

commute with each other, but each block does not commute with itself. These two concepts together achieve the goal of completeness and conciseness of commutativity specification.

Domain of Concurrency. A COMMSET aggregates a set of code blocks that read and modify shared program state. The members are executed concurrently in a larger parallelization scope, with atomicity of each member of the COMMSET guaranteed by automatic insertion of appropriate synchronization primitives. In this sense, COMMSET plays the role of a "concurrency domain," with updates to the shared mutable state being done in arbitrary order. However, the execution orders of members of a COMMSET with respect to the rest of code (sequential or other COMMSETs) are determined by flow dependences present in sequential code. These properties ensure that execution runs of a COMMSET program are serializable.

Non-transitivity and Multiple Memberships. Semantic commutativity is intransitive. Given three functions `f`, `g`, and `h` where two pairs (`f`, `g`), (`f`, `h`) semantically commute, the commuting behavior of (`g`, `h`) cannot be automatically inferred without knowing the semantics of state shared exclusively between `g` and `h`. Allowing code blocks to be members of multiple COMMSETs enables expression of either behavior. Programmers may create a single COMMSET with three members or create two COMMSETs with two members each, depending on the intended semantics. Two code blocks commute if they are both members of *at least* one COMMSET.

Commutative Blocks and Context Sensitivity. In many programs that access generic libraries, commutativity depends on the client code's context. The COMMSET construct is flexible enough to allow for commutativity assertions at either interface level or in client code. It also allows arbitrary structured code blocks to commute with other COMMSET members, which can either be functions or structured code blocks themselves. Functions

belonging to a module can export optional commuting behaviors of code blocks in their body by using named block arguments at their interface, without requiring any code refactoring. The client code can choose to enable the commuting behavior of the named code blocks at its call site, based on its context. For instance, the `mdfile` function in Figure 4.1 that has a code block containing calls to `fread` may expose the commuting behavior of this block at its interface via the use of a named block argument *READB*. A client which does not care about the order of `fread` calls can add *READB* to a COMMSET optionally at its call site, while clients requiring sequential order can ignore the named block argument. Optional COMMSET specifications do not require any changes to the existing interface arguments or its implementation.

Predicated Commutative Set. The COMMSET primitive can be predicated on either interface arguments or on variables in a client's program state. A predicate is a C expression associated with the COMMSET and evaluates to a Boolean value when given arguments corresponding to any two members of the COMMSET. The predicated expression is expected to be pure, i.e. it should return the same value when invoked with the same arguments. A pure predicate expression always gives a deterministic answer for deciding commutativity relations between two functions. The two members commute if the predicate evaluates to true when its arguments are bound to values of actual arguments supplied at the point where the members are invoked in sequential code.

Orthogonality to Parallelism Form. Semantic commutativity relations expressed using COMMSET are independent of the specific form of parallelism (data/pipeline/task) that are exploited by the associated compiler technology. Once COMMSET annotations are added to sections of a sequential program, the same program is amenable to different styles of parallelization. In other words, a single COMMSET application can express commutativity

relations between (static) lexical blocks of code and dynamic instances of a single code block. The former implicitly enables pipeline parallelism while the latter enables data and pipeline parallelism.

Well-defined CommSet members. The structure of COMMSET members has to obey certain conditions to ensure well-defined semantics in a parallel setting, especially when used in C/C++ programs that allow for various unstructured and non-local control flows. The conditions for ensuring well-defined commutative semantics between members of a COMMSET are: (a) The control flow constructs within each code block member should be local or structured, i.e. the block does not include operations like `longjmp, setjmp`, etc. Statements like `break` and `continue` should have their respective parent structures within the commutative block. (b) There should not be a transitive call from one code block to another in the same COMMSET. Removing such call paths between COMMSET members not only avoids the ambiguity in commutativity relation defined between a caller and a callee but also simplifies reasoning about deadlock freedom in the parallelized code. Both these conditions are checked by the compiler.

Well-formedness of CommSets. The concept of well-definedness can be extended from code blocks in a single COMMSET to multiple COMMSETs by defining a COMMSET graph as follows: A COMMSET *graph* is defined as a graph where there is a unique node for each COMMSET in the program, and there exists an edge from a node S_1 to another node S_2, if there is a transitive call in the program from a member in COMMSET S_1 to a member in COMMSET S_2. A set of COMMSETs is defined to be *well-formed* if each COMMSET has well-defined members and there is no cycle in the COMMSET graph. Our parallelization system guarantees deadlock freedom in the parallelized program if the only form of parallelism in the input program is implicit and is expressed through a well-formed set of

CommSet Global Declarations	`#pragma` **`CommSetDecl`**`(CSet,` **`SELF`**`|`**`Group`**`)`
	
	`#pragma` **`CommSetPredicate`**`(CSet,`
	`(`x_1`, ..., `x_n`), (`y_1`, ..., `y_n`),`
	`Pred(`x_1`, ..., `x_n`, `y_1`, ..., `y_n`))`
	
	`#pragma` **`CommSetNoSync`**`(CSet)`

CommSet Instance Declarations	`#pragma` **`CommSet`**`(CSets)`
	`|`**`CommSetNamedArg`**`(Blocks)`
	$type_{return}$ `function-name(`t_1 x_1`, ..., `t_n x_n`);`
	
	`#pragma` **`CommSet`**`(CSets)`
	`|`**`CommSetNamedBlock`**`(`B_{name}`)`
	`{`
	`Structured code region`
	`}`
	
	`#pragma` **`CommSetNamedArgAdd`**`(BPairs)`
	`retval = function-name(`x_1`, ...,`x_n`);`

CommSet List	`CSets ::= PredCSet | PredCSet, CSets`
	`PredCSet ::= CSet | CSet(CVars)`
	$CVars ::= C_{var} \mid C_{var}$`, Cvars`
	`CSet ::=` **`SELF`** `|` Set_{id}
	$BPairs ::= B_{id}$`(CSets) |`B_{id}`(CSets), BPairs`
	$Blocks ::= B_{id} \mid B_{id}$`, Blocks`

Figure 4.4: COMMSET Syntax

COMMSETs. This guarantee holds when either pipeline or data parallelism is extracted and for both pessimistic and optimistic synchronization mechanisms (see Section 4.5.6).

4.4 Syntax

This section describes the semantics of various COMMSET features and the syntax of the COMMSET primitives. The COMMSET extensions are expressed using `pragma` directives in the sequential program. `pragma` directives were chosen because a program with well-defined sequential semantics is obtained when they are elided. Programs with COMMSET annotations can also be compiled without any change by a standard C/C++ compiler that does not understand COMMSET semantics.

Global Declarations. The name of a COMMSET is indicative of its type. By default, the SELF keyword refers to a Self COMMSET, while COMMSETs with other names are Group COMMSETs. To allow for predication of Self COMMSETs, explicit type declaration can be used. The COMMSETDECL primitive allows for declaration of COMMSETs with arbitrary names at global scope. The COMMSETPREDICATE primitive is used to associate a predicate with a COMMSET and is declared at global scope. The primitive takes as arguments: (a) the name of the COMMSET that is predicated, (b) a pair of parameter lists, and (c) a C expression that represents the predicate. Each parameter list represents the subset of program state that decides the commuting behavior of a pair of COMMSET members, when they are executed in two different parallel execution contexts. The parameters in the lists are bound to either a commutative function's arguments, or to variables in the client's program state that are live at the beginning of a structured commutative code block. The C expression computes a Boolean value using the variables in the parameter list and returns true if a pair of COMMSET members commute when invoked with the appropriate arguments. By default, COMMSET members are automatically synchronized when their source code is available to the parallelizing compiler. A programmer can optionally specify that a COMMSET does not need compiler inserted synchronization using COMMSETNOSYNC. The primitive is applied to COMMSETs whose members belong to a thread-safe library which has been separately compiled and whose source is unavailable.

Instance Declarations and CommSet List. A code block can be declared a member of a list of COMMSETs by using the COMMSET directive. Such instance declarations can be applied either at a function interface, or at any point in a loop or a function for adding an arbitrary structured code block (a compound statement in C/C++) to a COMMSET. Both compound statements and functions are treated in the same way as far as reasoning about commutativity is concerned. In the case of predicated COMMSETs in the COMMSET list, the

actual arguments for the COMMSETPREDICATE are supplied at the instance declaration. For function members of a COMMSET, the actual arguments are a list of parameter declarations, while for compound statements, the actual arguments are a set of variables with primitive type that have a well-defined value at the beginning of the compound statement. Optionally commuting compound statements can be given a name by enclosing the statements within COMMSETNAMEDBLOCK directive. A function containing such a named block can expose the commuting option to client code using COMMSETNAMEDARG at its interface declaration. The client code that invokes the function can enable the commuting behavior of the named block by adding it to a COMMSET using the COMMSETNAMEDARGADD directive at its call site.

4.4.1 Example

Figure 4.1 shows the implicitly parallel program obtained by extending md5sum with COMMSET primitives. The code blocks *B*, *H*, *I* enclosing the file operations are added to a *Group* COMMSET *FSET* using annotations *5*, *7*, and *8*. Each code block is also added to its own *Self* COMMSET. *FSET* is predicated on the loop induction variable's value, using a COMMSETPREDICATE expression (*3*) to indicate that each of the file operations commute with each other on separate iterations. The block containing `fread` call is named *READB* (*10*) and exported by `mdfile` using the COMMSETNAMEDARG directive at its interface (*9*). The client code adds the named block to its own *Self* set (declared as *SSET* in *2*) using the COMMSETNAMEDARGADD directive at *6*. *SSET* is predicated on the outer loop induction variable to prevent commuting across inner loop invocations (*4*). A deterministic output can be obtained by omitting *SELF* from annotation *7*.

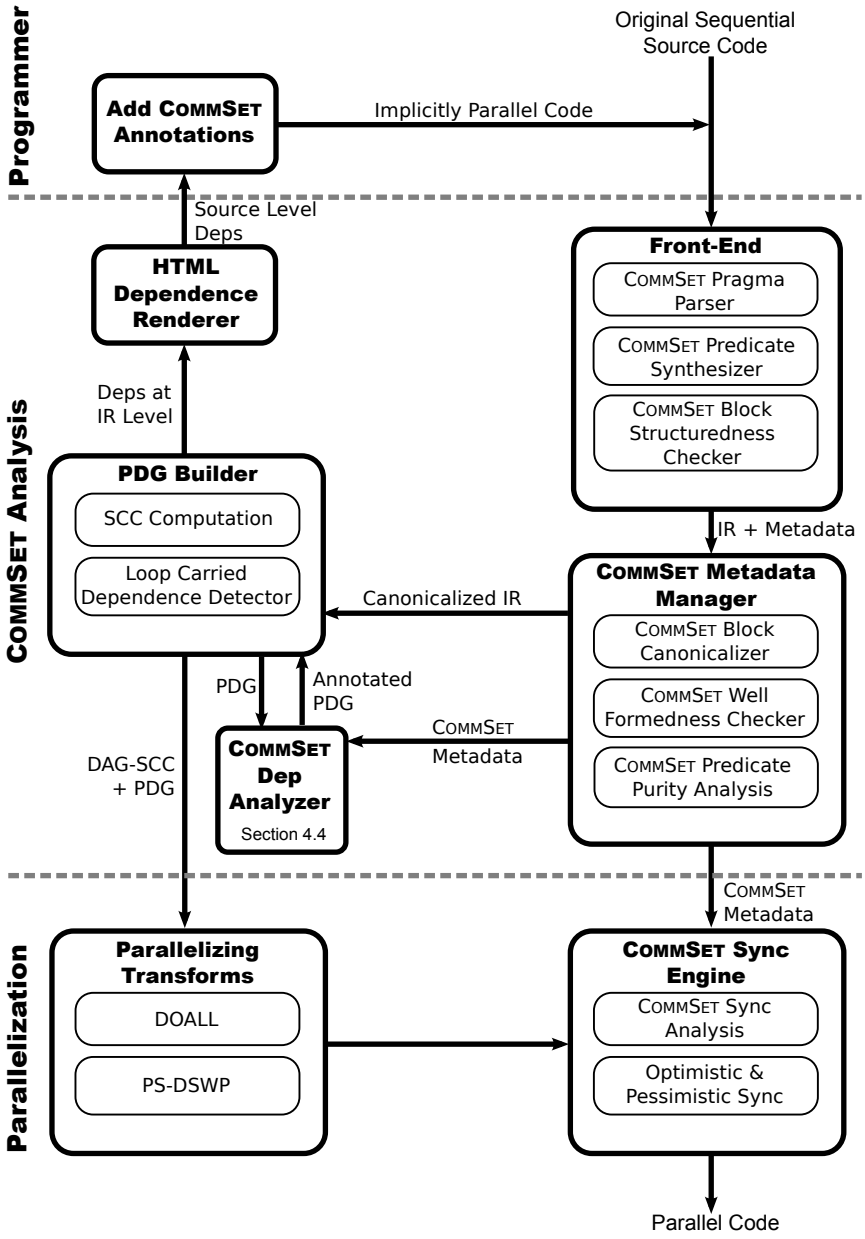


Figure 4.5: COMMSET Parallelization Workflow

4.5 Compiler Implementation

We built an end-to-end implementation of COMMSET within a parallelizing compiler. The compiler is an extension of the clang/LLVM framework [130]. Figure 4.5 shows the parallelization workflow. The parallelization focuses on hot loops in the program identified

via runtime profiling. The PDG for the hottest loop is constructed over the LLVM IR, with each node representing an instruction in the IR. The memory flow dependences in the PDG that inhibit parallelization are displayed at source level to the programmer, who inserts COMMSET primitives and presents the program back to the compiler. The subsequent compiler passes analyze and transform this program to generate different versions of parallelized code.

4.5.1 Frontend

The COMMSET parser in the frontend parses and checks the syntax of all COMMSET directives, and synthesizes a C function for every COMMSETPREDICATE. The predicate function computes the value of the C expression specified in the directive. The argument types for the function are automatically inferred by binding the parameters in COMMSETPREDICATE to the COMMSET instances. Type mismatch errors between arguments of different COMMSET instances are also detected. Commutative blocks are checked for enclosing non-local control flow by a top-down traversal of the abstract syntax tree (AST) starting at the node corresponding to the particular commutative block. Finally, global COMMSET meta-data is annotated at the module level, while COMMSET instance data is annotated on individual compound statement or function AST nodes. This meta-data is automatically conveyed to the backend during the lowering phase.

4.5.2 CommSet Metadata Manager

In the backend, the COMMSET meta-data is an abstraction over the low-level IR constructs, instead of the AST nodes. The COMMSET Metadata Manager processes and maintains a meta-data store for all COMMSET instances and declarations, and answers queries posed by subsequent compiler passes. The first pass of the manager canonicalizes each commutative

compound statement, now a structured region (set of basic blocks) within the control flow graph, by extracting the region into its own function. Nested commutative regions are extracted correctly by a post-order traversal on the control flow graph (CFG). The extraction process ensures that arguments specified at a COMMSET instance declaration are parameters to the newly created function. At the end of this pass, all the members of a COMMSET are functions. Call sites enabling optionally commutative named code blocks are inlined to clone the call path from the enabling function call to the COMMSETNAMEDBLOCK declaration. A robust implementation can avoid potential code explosion by automatically extending the interface signature to take in additional arguments for optional commuting blocks. Next, each COMMSET is checked for well-formedness using reachability and cycle detection algorithms on the call graph and the COMMSET graph respectively. The COMMSETPREDICATE functions are tested for purity by inspection of its body.

4.5.3 PDG Builder

The PDG builder constructs the PDG over the LLVM IR instructions for the target loop using well-known algorithms [78]. A loop carried dependence detector module annotates dependence edges as being loop carried whenever the source and/or destination nodes read and update shared memory state. Memory dependence edges are constructed by relying on a variety of alias analysis techniques [153] composed together, including those that support loop, field, calling-context, shape and type sensitivity. In addition, to disprove certain spurious memory dependences involving external functions, our alias analysis implementation was enhanced with custom aliasing rules (see Table 6.2 in Chapter 6). However, it should be noted that COMMSET annotations were applied only in instances where there is a real memory flow dependence due to the sequential programming model.

Algorithm 1: *CommSetDepAnalysis*

```
1  foreach edge e ∈ PDG do
2      let n1 = src(e); let n2 = dst(e);
3      if typeOf(n1) ≠ Call ∨ typeOf(n2) ≠ Call then
4          continue
5      end
6      let Fn(n1) = f(x1, ..., xn) and Fn(n2) = g(y1, ..., yn);;
7      let S_in = CommSets(f) ∩ CommSets(g);
8      foreach C_s ∈ S_in do
9          if not Predicated(C_s) then
10             Annotate(e, PDG, uco);
11         end
12         else
13             let f_p = PredicateFn(C_s);
14             let args1 = CommSetArgs(C_s, f);
15             let args2 = CommSetArgs(C_s, g);
16             let fargs = FormalArgs(f_p);
17             for i = 0 to | args1 − 1 | do
18                 let x1 = args1(i) ; let x2 = args2(i);
19                 let y1 = fargs(2 ∗ i) ; let y2 = fargs(2 ∗ i + 1);
20                 Assert(x1 = y1); Assert(x2 = y2);
21             end
22             if LoopCarried(e) then
23                 Assert(i1 ≠ i2); // induction variable;
24                 r = SymInterpret(Body(f_p), true);
25                 if (r = true) and (Dom(n2, n1)) then
26                     Annotate(e, PDG, uco);
27                 end
28                 else if (r = true) then
29                     Annotate(e, PDG, ico);
30                 end
31             end
32             else
33                 r = SymInterpret(Body(f_p), true);
34                 if (r = true) then
35                     Annotate(e, PDG, uco);
36                 end
37             end
38         end
39     end
40 end
```

4.5.4 CommSet Dependence Analyzer

The COMMSET Dependence Analyzer (Algorithm 1) uses the COMMSET metadata to annotate memory dependence edges as being either unconditionally commutative (uco) or inter-iteration commutative (ico). Figure 4.2 shows the PDG edges for md5sum annotated with commutativity properties along with the corresponding source annotations. For every memory dependence edge in the PDG, if there exists an unpredicated COMMSET of which both the source and destination's target functions are members, the edge is annotated as uco

(Lines 9-11). For a predicated COMMSET, the actual arguments of the target functions at their call sites are bound to corresponding formal parameters of the COMMSETPREDICATE function (Lines 17-19). The body of the predicate function is then symbolically interpreted to prove that it always returns *true*, given the inequality assertions about induction variable values on separate iterations (Lines 21-22). If the interpreter returns true for the current pair of COMMSET instances, the edge is annotated with a commutativity property as follows: A loop carried dependence is annotated as uco if the destination node of the PDG edge dominates the source node in the CFG (Lines 23-34), otherwise it is annotated as ico (Lines 26-27). An intra-iteration dependence edge is always annotated as uco if the predicate is proven to be true (Lines 32-34). Once the commutative annotations are added to the PDG, the PDG builder is invoked again to identify strongly connected components (SCC) [118]. The directed acyclic graph of SCCs (DAG-SCC) thus obtained forms the basis of DSWP family of algorithms [157].

4.5.5 Parallelizing Transforms

The next step runs the DOALL and PS-DSWP parallelizing transforms, which automatically partition the PDG onto multiple threads for extracting maximal data and pipelined parallelism respectively. For all the parallelizing transforms, the *ico* edges are treated as intra-iteration dependence edges, while *uco* edges are treated as non-existent edges in the PDG. The DOALL transform tests the PDG for absence of inter-iteration dependencies, and statically schedules a set of iterations to run in parallel on multiple threads. The DSWP family of transforms partition the DAG-SCC into a sequence of pipeline stages, using profile data to obtain a balanced pipeline. The DSWP algorithm [158] only generates sequential stages, while the PS-DSWP algorithm [173] can replicate a stage with no loop carried SCCs to run in parallel on multiple threads. Dependences between stages are communicated via lock-free queues in software. Together, the *uco* and *ico* annotations on the PDG

enable DOALL, DSWP, and PS-DSWP transforms when previously they were not applicable. Currently, the compiler generates one of each (DSWP, PS-DSWP, and DOALL) schedule whenever applicable, with a corresponding performance estimate. A production quality compiler would typically use heuristics to select the optimal across all parallelization schemes.

4.5.6 CommSet Synchronization Engine

This step automatically inserts synchronization primitives to ensure atomicity of COMMSET members with respect to each other, taking multiple COMMSET memberships into account (Algorithm 2). The compiler generates a separate parallel version for every synchronization method used. Currently three synchronization modes are supported: optimistic (via Intel's transactional memory (TM) runtime [218]), pessimistic (mutex and spin locks) and lib (well known thread safe libraries or programmer specified synchronization safety for a COMMSET). Initially, the algorithm assigns a unique rank to each COMMSET which determines the global order of lock acquires and releases. The next step determines the set of potential synchronization mechanisms that apply to a COMMSET. Synchronization primitives are inserted for each member of a COMMSET by taking into account the other COMMSETs it is a part of. In the case of TM, a new version of the member wrapped around transactional constructs is generated. For the lock based synchronizations, lock acquires and releases are inserted according to the assigned global rank order. The acyclic communication primitives that use the lock free queues together with the synchronization algorithm maintain the following two invariants required to ensure deadlock freedom [132]:

- A blocking consume on a lock-free queue is matched by a produce with the same control flow conditions [157].
- Each thread performs lock and unlock operations in accordance with a global order

Algorithm 2: *CommSetAutoSynch*

Input: P_{in}: Unsynchronized Parallel Program, P_T: Thread Partition, S_{CS}: Set of Commutative Sets
Output: P_{out}: Synchronized Parallel Code

let $Ctr_{rank} = 0$
for $C_s \in S_{CS}$ **do**
  $Rank(C_s) = Ctr_{rank}$
  $Ctr_{rank} = Ctr_{rank} + 1$
end
foreach $C_s \in S_{CS}$ **do**
  foreach $f \in C_s$ **do**
    if $Callees(f) = \emptyset$ **and not** $Declaration(f)$ **then**
      $Synch_{attr}(S) = \{TM\}$
    end
  end
  if $\bigcap_{f \in C_s} ModRef(f) = \{x \mid x \in AbsLoc_{Lib}\}$ **then**
    $Synch_{attr}(C_s) = \{NOSYNC\}$
  end
  if $NOSYNC \notin Synch_{attr}(C_s)$ **then**
    $Synch_{attr}(C_s) \bigcup = \{SPIN, MUTEX\}$
  end
  if *(*$\exists\ f \in C_s$ *which is part of a parallel stage in* P_T*)* **then**
    $S_{cands} \bigcup = C_s$
  end
end
for $f \in$ *partition in* P_T **do**
  if $f \notin S_{cands}$ **then**
    continue
  end
  let $S_f = \{C_s \mid \exists C_s \in S_{cands}\ such\ that\ f \in C_s\}$
  $Synch_{int}(f) = \bigcap_{C_s \in S_f} Synch_{attr}(C_s)$
  if $Synch_{int}(f) = \emptyset$ **then**
    raise $InCompatibleSynchException$
  end
  if $S_f = \{NOSYNC\}$ **then**
    pass
  end
  if $TM \in S_f$ **then**
    Generate a version of f wrapped with `__tm_atomic`
  end
  if $MUTEX \in S_f$ **then**
    Create a mutex lock for each $C_s \in S_f$ if not already created
    $Lock_{seq}$ = Order locks associated with each $C_s \in S_f$ using $Rank(C_s)$
    Generate a version $f^{'}$ of f
    for $i = 1\ to\ \mid Lock_{seq} \mid$ **do**
      Insert $pthread\_mutex\_lock(\&lock_i)$ in the start block of $f^{'}$
      Insert $pthread\_mutex\_unlock(\&lock_i)$ at the exit block of $f^{'}$
    end
  end
  `// Repeat above step for spin locks`
end

After this step, the parallelized code is optimized using various scalar optimization passes and is ready to run.

Chapter 5

Weakly Consistent Data Structures

Search and optimization problems play an important role in many modern-day scientific applications. These problems are typically combinatorial in nature, having search spaces that are prohibitively expensive to explore exhaustively. As a result, many real-world algorithmic implementations for these problems employ various heuristic techniques. One common technique uses auxiliary data structures: as the main loop iteratively explores the search space, new facts are recorded in the data structure. These facts are later retrieved by subsequent iterations to prune or guide the search. Examples of such data structures include kernel caches in support vector machines [108], learned clause databases in SAT solvers [76], elite sets in genetic algorithms [140], cuts in mixed integer linear programs [170], "wisdom" data structures in FFT planners [80], and transposition tables in chess programming [187]. This technique greatly improves the running times of such search algorithms.

The emergence of multicore architectures onto mainstream computing presents a tremendous opportunity to parallelize many search and optimization algorithms. Explicitly parallelizing search loops by synchronizing accesses to the auxiliary data structure is one way of

Programming Model	Concept (Section 5.3)			
	Out of Order	Partial View	Weak Deletion	Annotation Type
N-way [54]	✓	✓	×	I
Semantic Commutativity [40, 179]	✓	×	×	I
Galois [128]	✓	×	×	I
Cilk++ hyperobjects [79]	✓	×	×	E
ALTER [203]	✓	✓	×	I
WEAKC	✓	✓	✓	I

Table 5.1: Conceptual comparison between WEAKC and related parallelization frameworks (I: Implicitly Parallel, E: Explicitly Parallel)

optimizing such programs. This is however a slow, manual and error-prone process which results in performance-unportable and often sub-optimal programs.

5.1 Prior Work and Limitations

Recent work demonstrates how high-level semantic programming extensions can facilitate parallelization. Semantic commutativity extensions [40, 128, 179] allow functions that atomically access shared state to execute out-of-order concurrently, breaking flow dependences across function invocations. Cilk++ hyperobjects [79] and the N-way programming model [54] provide programming support to implicitly privatize data for local use within each parallel task before starting execution, and provide join semantics upon completion of parallel execution. ALTER [203] presents an optimistic execution model allowing parallel tasks to read stale values of shared data as long as they do not write to the same locations.

Programs accessing auxiliary data structures can be parallelized by the methods described above, but each has its pitfalls: (a) synchronizing *every* access to a shared data structure as implied by semantic commutativity is expensive, given the high frequency of access to these data structures; (b) complete privatization of auxiliary data structures within each parallel task without communicating any auxiliary data can adversely affect the convergence times of search loops by providing fewer pruning opportunities; and (c) with the

Programming Model	Parallel Implementation (Sections 5.5 and 5.6)		
	Parallelization Driver	Sharing	Synchronization
N-way [54]	Runtime	None	Static
Semantic Commutativity [40, 179]	Compiler	All	Static
Galois [128]	Runtime	All	Static
Cilk++ hyperobjects [79]	Programmer	None	Static
ALTER [203]	Compiler	All	Static
WEAKC	Compiler & Runtime	Sparse	Static/Adaptive

Table 5.2: Comparison between the WEAKC implementation and the other related parallelization frameworks

ALTER model, conflicts based on writes to memory would occur frequently due to updates to auxiliary data structures.

WEAKC is based on the following insight: auxiliary data structures present a much stronger sequential semantics than required by the programs using them. Such programs often continue to function correctly even when queries to the data structure return stale values, or when inserted values are absent. This weaker semantics can be leveraged to break dependences between data structure operations, potentially facilitating parallelization of the main search loops. Once parallelized, it is important to optimize the tradeoff between increased parallelism and potential delay in algorithmic convergence. Frequent communication of auxiliary data between parallel tasks can lead to shorter algorithmic convergence at the cost of synchronization overheads, and conversely little or no communication of auxiliary data may prolong algorithmic convergence while reducing synchronization overhead.

This chapter presents WEAKC, a framework for efficiently parallelizing search loops that access auxiliary data structures. WEAKC consists of:

- semantic language extensions for weakening consistency of data structures, which expose parallelism in code that uses them
- a compiler that parallelizes search loops;

- a runtime system that adaptively optimizes parallel configurations of auxiliary data structures.

Tables 5.1 and 5.2 compare WEAKC with frameworks that leverage relaxed semantics for parallelization. The main advantage of WEAKC stems from expressing a much weaker semantics than any of the other frameworks, and having the WEAKC parallelization framework leverage this weaker semantics to realize an adaptively synchronized parallel scheme, as opposed to static parallel schemes realized by others.

5.2 Motivating Example

WEAKC is motivated by the boolean satisfiability problem (SAT), a well-known search problem with many important applications. A SAT solver attempts to solve a given boolean formula by assigning values to variables such that all clauses are satisfied, or determine that no such assignment exists. Figures 5.1 and 5.2 show main parts of a SAT solver's [76] C++ implementation.

The main search loop of a SAT solver selects an unassigned variable, set its value, and recursively propagates this assignment until either all variables are assigned (thereby completing a solution) or a constraint becomes conflicting under the current assignment. In case of a conflict, an analysis procedure learns a clause implying the conflict, records it in an auxiliary data structure (Line 11), and applies backtracking.

Learnt clauses help subsequent iterations prune their search space. When learning is disabled, we found an average slowdown of 8.15x when running a sequential SAT solver across several workloads (Figure 5.3). However, while more learning implies more pruning opportunities, traversals of the auxiliary data structure (Line 4) can slow down considerably if its size grows excessively, and hence the SAT solver loop periodically removes "useless" learnt clauses (Line 21).

```
1 lbool Solver::search(int nof_learnts) {
2  model.clear();
3  while (1) {
4   // propagate() accesses learnts[i]
5   Constr confl = propagate();
6   if (confl != NULL) { // Conflict
7    if (decisionLevel() == root_level)
8      return False;
9    learnt_clause =analyze(confl, backtrack_level);
10   cancelUntil(max(backtrack_level, root_level));
11   learnts.push(learnt_clause);
12   decayActivities ();
13   ... // backtrack
14  }
15  else { // No conflict
16   ...
17   int n = learnts.size() - nAssigns();
18   if (n >= nof_learnts) {
19     // Reduce the set of learnt clauses
20     for (int i=0; i < 2*n; ++i)
21       learnts.remove(learnts[i]);
22   }
23   if (nAssigns () == nVars ()) {
24     // Model found:
25     model.growTo(nVars ());
26     ...
27     cancelUntil (root_level);
28     return True;
29   } else {
30     // New variable decision
31     lit p = lit (order.select());
32     assume (p);
33   }
34  }
35 }
36 }
```

Figure 5.1: The SAT main search loop skeleton that accesses and modifies `learnts` database

Considering the intractable nature of the SAT problem and the increasing availability of multicore on desktops and servers, parallelizing such search loops has the potential to drastically reduce search times. One common approach to parallelize search and optimization algorithms is *multisearch* [198]. In this approach multiple parallel workers search in distinct regions of the search space, until some worker finds a solution. Convergence times are

```
37 class Solver {
38   protected:
39 #pragma WeakC(SET, L)
40    vec<CRef> learnts;
41 #pragma WeakC(ALLOCATOR, A)
42    ClauseAllocator ca;
43    ...
44    void attachClause     (CRef cr);
45 };
46
47 template<class T> class vec {
48   public:
49 #pragma WeakCI(L, SZ);
50     int size(void) const;
51 #pragma WeakCI(L, ILU)
52     T& operator [](int index);
53 #pragma WeakCI(L, INS)
54     void push(const T& elem);
55 #pragma WeakCI(L, DEL)
56     void remove(const T& elem);
57     ...
58 };
59
60 class ClauseAllocator : public RegionAllocator
61 {
62   public:
63 #pragma WeakCI(A, ALLOC)
64    void reloc(CRef& cr, ClauseAllocator& to);
65 #pragma WeakCI(A, DEALLOC)
66    void free(CRef cid);
67    ...
68 };
71 #pragma WeakCI(L, PROG)
72 double Solver::progressEstimate() const {
73    return  (pow(F,i)* (end-beg))/ nVars();
74 }
```

Figure 5.2: SAT class declarations with WEAKC annotations

greatly improved when clauses learnt by each worker are shared with other workers [91]. However, manually parallelizing a SAT search loop using explicit parallel programming requires considerable effort. Auto-parallelizing tools can relieve programmers from this effort, but are constrained by the need to respect sequential semantics. In our example, read accesses, inserts and deletes into the `learnts` data structure within the search loop are inter-dependent (Lines 4, 11 and 21), inhibiting parallelization.

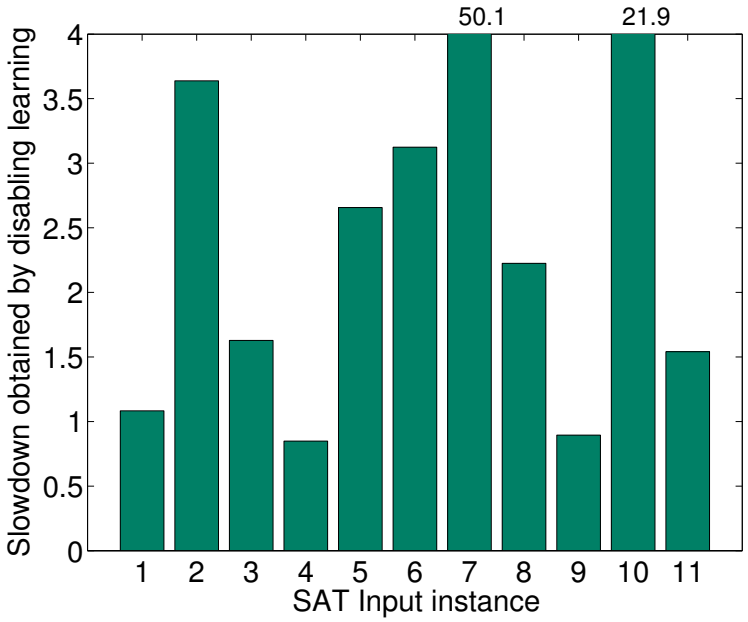


Figure 5.3: Impact of disabling learning on sequential SAT execution time

The use of `learnts` data structure in the context of SAT solvers, however, requires much weaker consistency than imposed by a sequential programming model. In particular, the SAT search loop will function correctly even when reads from `learnts` only return a partial set of clauses inserted so far, or when deleted clauses persist. This weaker semantics of `learnts` can be used to break dependences between data structure operations and facilitate parallelization. Programming extensions can express this weaker semantics, enabling transformation tools to parallelize the loop without sacrificing ease of sequential programming. Note that a SAT solver using `learnts` with weaker consistency property may explore the search space differently from a sequential search, which may result in a different satisfying assignment on each run.

Recent work on semantic extensions to the sequential programming model for parallelism impose much stronger requirements on the `learnts` data structure operations than required by SAT solvers. In these solutions, all inserts into `learnts` have to eventually succeed [203] and operations either access one shared `learnt` data structure which is

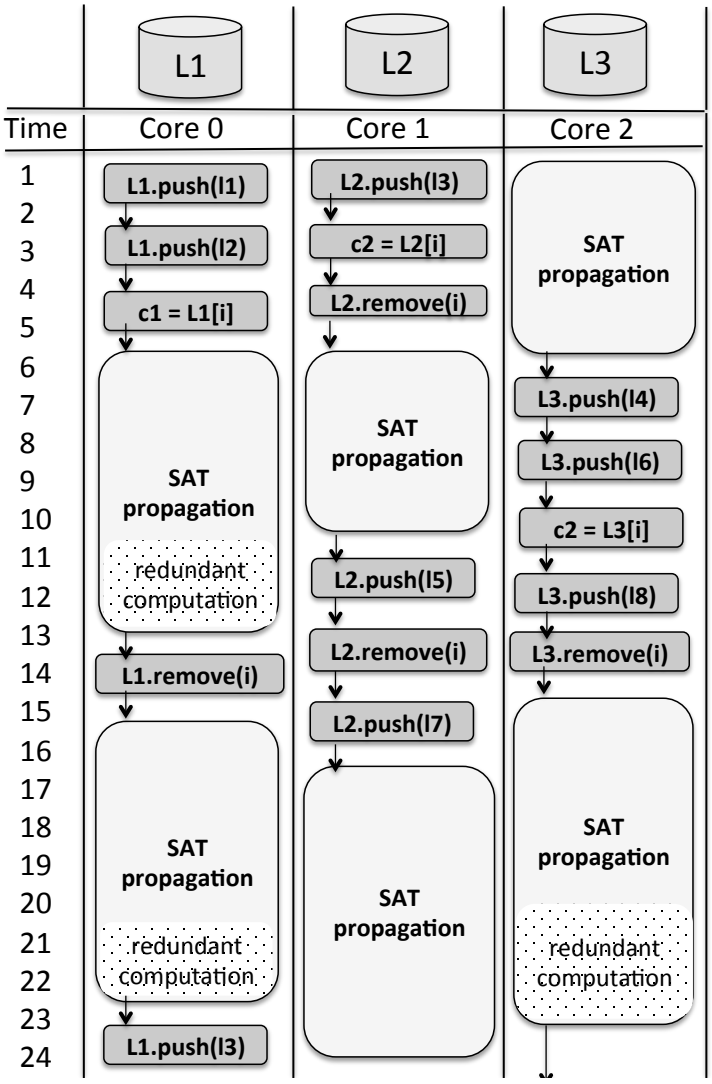


Figure 5.4: Parallel execution timeline of Privatization (N-Way)

always synchronized [40, 128, 179] or operate on multiple private copies with no clause sharing [54, 79]. Choosing "all" or "nothing" synchronization strategies for `learnts` data structure can adversely affect parallel performance: using fine-grain synchronization mechanisms to completely share every clause learnt amongst the workers can lead to increased overheads while having private copies with no sharing can increase the convergence time due to redundant computation.

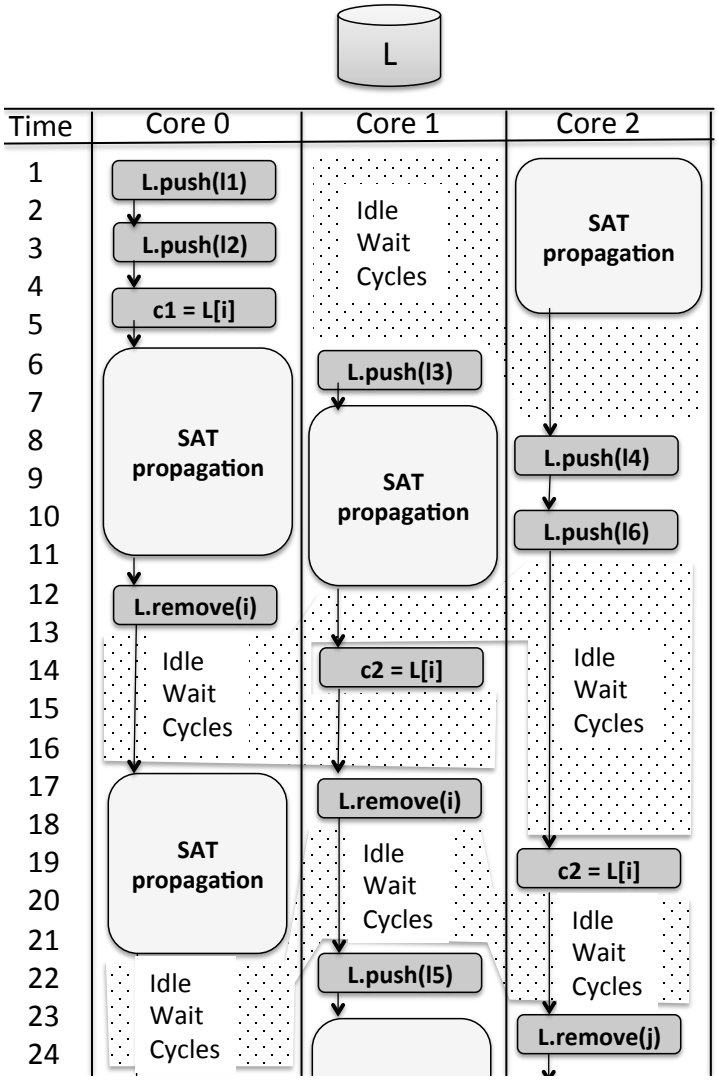


Figure 5.5: Parallel execution timeline of Complete-Sharing (Semantic Commutativity)

Instead, *sparse sharing*, a hybrid synchronization strategy that completely exploits the weak consistency property of `learnts` has much better performance characteristics compared to private and complete sharing strategies. In sparse sharing, each worker maintains a copy of the `learnts` data structure and *selectively synchronizes* with other workers by exchanging *subsets* of locally learnt clauses at certain intervals, balancing the tradeoff between synchronization overhead and algorithmic convergence. Figures 5.4, 5.5, and 5.6

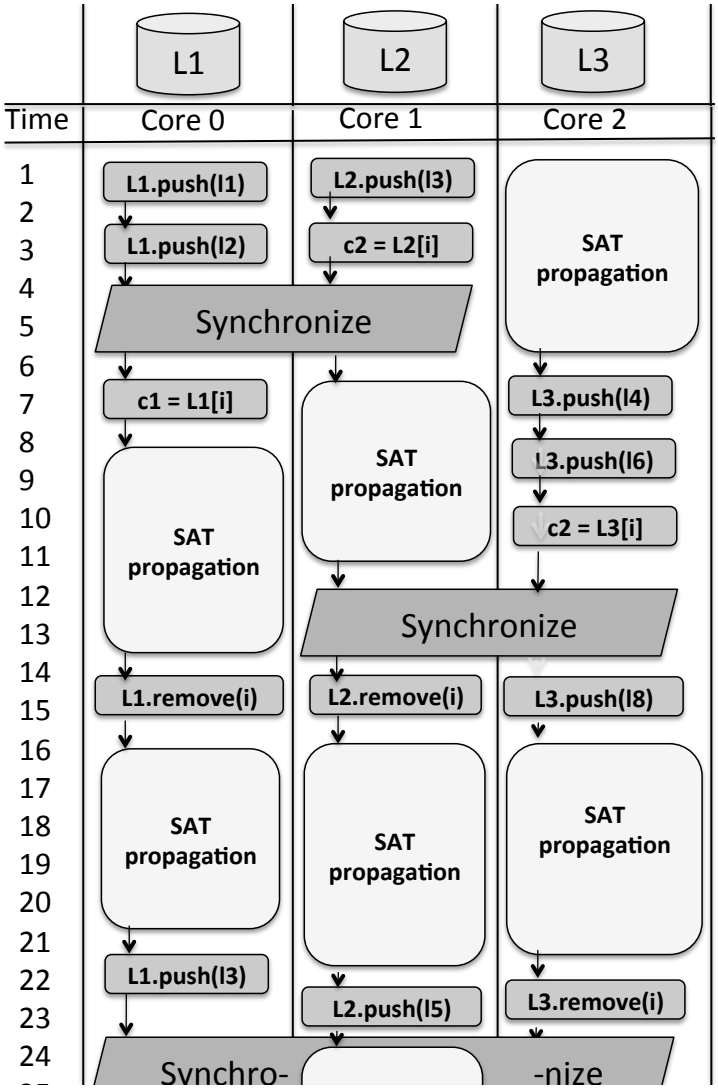


Figure 5.6: Parallel execution timeline of WEAKC

compare the timeline of parallel multisearch using the three synchronization strategies outlined above.

The running times of the SAT solver, similar to many search and optimization algorithms, varies widely depending on characteristics of their input. Very often candidate programs exhibit phase changes in their computation [219]. In such scenarios adapting the synchronization strategy to the differing progress rates of parallel workers can accelerate convergence. For instance, during certain phases some workers might quickly learn facts

that could help other workers prune their search space considerably. Online adaptation tunes to this varying runtime behavior, effectively enabling workers to cooperatively find a solution faster than otherwise possible.

Varying the synchronization strategy is only one parameter exposed by the weakened semantics of the `learnts` data structure. Allowing inserted clauses to disappear opens new opportunities for efficient eviction mechanisms, and mechanisms for deciding which clauses to share, with whom and when. These parameters are important due to the tradeoff between the usefulness of the the `learnts` database and the time spent synchronizing it, which is proportional to its size. Given the dynamic nature of SAT search, adapting these parameters at runtime in response to the varying progress made by different workers (measured on Line 72 in our example) has the potential to accelerate search convergence.

5.3 Semantics

We describe the semantics of a weakly consistent data structure in terms of operations supported on the data structure through its interface. A *weakly consistent set* is defined as an abstract data structure that stores values, in no particular order, and may contain repeated values similar to multiset. The semantics of the operations supported by a weakly consistent set is as follows:

1. **Out of order mutation:** The insertion operations into a weakly consistent set can be executed in a different order from that of a sequential specification. Similarly, deletions of multiple elements can execute out of order with respect to each other and also with respect to earlier insertions of non-identical elements. Both these properties follow directly from a conventional set definition [96].

2. **Partial View:** The set supports lookup operations that may return only a subset of its contents, giving only a partial view of the data structure that may not reflect the latest

insertions. Each element returned by a lookup must, however, have been inserted at *some earlier* point in the execution of the program [1]. This partial view semantics corresponds to the concept of *weak references* [70] in managed languages like Java, where it is used for non-deterministic eviction of inserted elements allowing for early garbage collection. In the SAT example, a partial view of learnt clause set during propagation may only reduce pruning opportunities, but has no effect on program correctness.

3. **Weak Deletion:** Deletions into a weakly consistent data structure need not be persistent, with the effect of deletions lasting for a non-deterministic length of time. When combined with partial view, this semantics implies that between two lookup operations that are invoked after a deletion, certain elements not present in the first lookup may appear during the second lookup without any explicit insertion operations. In the SAT example, the weak deletion property safely applies to the learnt clause set due to the temporally-agnostic nature of learnt clauses: since a learnt clause remains globally true regardless of when it was learnt, it is safe to re-materialize deleted clauses.

The abstract semantics outlined above translates into concurrent semantics by adding one additional property: atomicity of individual operations. A weakly consistent set can have different concurrent implementations, each trading consistency for different degrees of parallelism. We describe three realizations of a weakly consistent set, two of which – privatization and complete sharing have been studied in the context of other parallel programming models. The third realization, titled sparse sharing, takes full advantage of weakened semantics for optimizing the tradeoff between consistency and parallelism, and is

[1] The terms *later* and *earlier* relate to a time-ordered sequence of operations invoked on the data structure as part of an execution history or trace

the one implemented by WEAKC. In the following, the term *client* refers to an independent execution context such as a thread or an OS process.

1. **Privatization** [54, 79]: A weakly consistent set can be implemented as a collection of multiple private replicas of a sequential data structure, one for each client. Operations invoked by a client operate on its private replica. There is no interference between clients and all operations are inherently atomic without any synchronization. Insertion and deletion operations on a single replica execute in sequential order while executing out of order with respect to other replica operations. In any time-ordered history of operations, lookups in a client only return elements inserted earlier into a local replica and not the remote ones, thus giving only a partial global view of the data structure. However, deletion operations are persistent: once an element is deleted in a replica, it will not be returned by a subsequent lookup on that replica without an explicit insertion of the same element.

2. **Complete Sharing** [40, 128, 179]: A weakly consistent set is realized by a single data structure that is shared amongst all clients. Operations performed on the data structure by clients are made atomic by wrapping them in critical sections and employing synchronization. This realization is implied by and implemented using semantic commutativity annotations on the operations of the data structure. In this method insertions and deletions of non-identical elements can occur out of order. Lookups present a complete view of the data structure for each client and return the most up-to-date, globally available contents of the set. Deletion of an element is persistent, with effects immediately visible across all clients.

3. **Sparse Sharing**: A third realization of a weakly consistent set combines the above two to take full advantage of the partial view and weak deletion property. This scheme uses a collection of replicas, one for each client, and non-deterministically

WEAKC Data Structure Declarations	`#pragma WeakC(dstype, id)`
	`  `$type_d$ `obj;`
	
	`dstype := SET | MAP | ALLOCATOR`

WEAKC Interface Declarations	`#pragma WeakCI(id, itype)`
	`  `$type_r$ `class::fn(`t_1 p_1`, ...);`
	
	`itype := INS | DEL | ILU | ALLOC`
	`SZ | DEALLOC | PROG | CMP`

Figure 5.7: WEAKC Syntax

switches between two sharing modes. In private mode, each operation is performed on the local replica without any synchronization. In sharing mode, insertions are propagated transparently to remote replicas while deletions continue to apply to the local replica only. In this model, lookups return a partial view of cumulative contents of all replicas: elements made visible to a client correspond to a mix of insertions performed locally and remotely at some earlier point in execution. Deletions are weak: an element deleted in a local replica can reappear if it has been propagated to a remote replica prior to deletion and is propagated back to the local replica transparently at some point after deletion.

The weak consistency concept easily extends to other sequential data structures as well. In particular, a *weakly consistent map* is very useful for applications performing associative lookups of memoized data. The use of kernel caches in SVMs [108], "wisdom" data structures in FFT planners [80], and transposition tables in alpha-beta search [187] are instances of such weakly consistent maps.

5.4 Syntax

We describe a WEAKC data structure using a standard abstract interface. This interface corresponds to that of an indexed set class with member functions for insertion, deletion,

indexed search and querying the size of the data structure. WEAKC extensions are specified as `pragma` directives within a client's sequential program [2]. `pragma`s are used for two reasons: first, arbitrary data structure implementations within a client can be regarded as weakly consistent by annotating relevant accessor and mutator interfaces using WEAKC's `pragma`s without any major re-factoring. Second, use of `pragma`s preserves the sequential semantics of the program when the annotations are elided and allows programs with WEAKC's annotations to be compiled unmodified by C++ compilers that are unaware of WEAKC semantics.

Figure 5.7 shows the syntax of WEAKC directives which include two parts: extensions to annotate a data structure with a WEAKC type at its instantiation site, and for declaring the data structure's accessors and mutators as part of the WEAKC interface. The two main WEAKC data structures are the weakly consistent set (SET) and map (MAP), with corresponding member functions for insertion (INS), deletion (DEL), indexed search (ILU), and querying the size of the data structure (SZ). Combining ILU and SZ gives a multi-element lookup functionality. An allocator (ALLOCATOR) class can be optionally associated with a weakly consistent data structure. This class is akin to allocators for C++ standard template library classes, and can be used when collections of pointers that rely on custom memory allocation are annotated as weakly consistent. The associated allocation (ALLOC) and deallocation (DEALLOC) methods are used by WEAKC to transparently orchestrate replication and sharing among parallel workers. Finally, the parent solver class containing a weakly consistent data structure can specify a member function that reports search progress (PROG) – a real valued measure between 0 and 1, and a member function

[2] WEAKC also provides a library of weakly consistent data structures based on C++ templates akin to STL that can be directly used

to compare the relative quality of weakly consistent element pairs (CMP). Both these functions serve as hints to the WEAKC runtime for optimizing sparse sharing. Figure 5.2 shows WEAKC directives applied to the SAT example.

5.5 The WEAKC Compiler

The WEAKC parallelization workflow is shown in Figure 5.8. The programmer first annotates a sequential program using `pragmas` at interface declarations and data structure instantiations. The annotated program is then fed into the WEAKC compiler which analyzes it to extract semantic information (in its frontend) and later introduces parallelism (in its backend). The resulting parallelized program executes in conjunction with the WEAKC runtime that performs online adaptation. Each phase is explained in detail in following subsections.

5.5.1 Frontend

The WEAKC compiler frontend is based on clang++ [129]. It includes a `pragma` parser, a type collector module within the compiler's semantic analysis, and a source-to-source rewriter consuming the output of the type collector module. The `pragma` parser records source level and abstract syntax tree (AST) level information about weakly consistent data structures and interface declarations, within in-memory data structures, and includes support for multiple `pragmas` associated with the same source entity.

The type collector module uses WEAKC information associated with AST nodes to correctly deduce fully qualified (including `namespace` information) source level type information of weakly consistent containers and the contained data's type from instantiation sites. Additionally, the type collector gathers and checks that the type information of all annotated interface declarations (including overloaded operators) to see if they conform

to declarations of the canonical weakly consistent data structure prototypes. Finally, the rewriter module adds and initializes additional data member fields into the parent class of weakly consistent types. In particular, it includes a unique identifier field for distinguishing between different object instances belonging to different parallel workers at runtime, to support an object based parallel execution model [114] within WEAKC.

The output of the rewriter is then translated by the frontend from C++ code into LLVM IR [129], titled "Augmented Source IR" in Figure 5.8. Meta-data describing WEAKC annotations is embedded within this IR destined for the backend. The frontend also creates runtime hooks — fully qualified types and names of WEAKC data structures and interface declarations that are used by the runtime instantiator to: (a) instantiate concrete types for abstract data types declared within the WEAKC runtime library; (in the form of `typedef` declarations); (b) synthesize functors for measuring progress of each solver; and (c) specialize the runtime's function templates with the concrete types and interface declarations of the client program. The output of the runtime instantiator is then translated into LLVM IR by feeding it into the frontend, producing a version of WEAKC runtime library specialized to the current client program.

5.5.2 Backend

The WEAKC compiler backend first profiles the augmented source IR using a standard loop profiler to identify hot loops. Search programs typically concentrate their execution on one main loop that iterates until convergence. The hot loop found is search loop's body, after identification using profile data, is marked for runtime instrumentation by subsequent backend passes. The WEAKC parallelizer injects parallelism into the IR by spawning threads at startup, replicating parent solver objects to introduce object based concurrency to perform search in parallel, granting ownership of each replica to a newly spawned thread, and starting parallel execution. These transformations are applied at the outermost loop level.

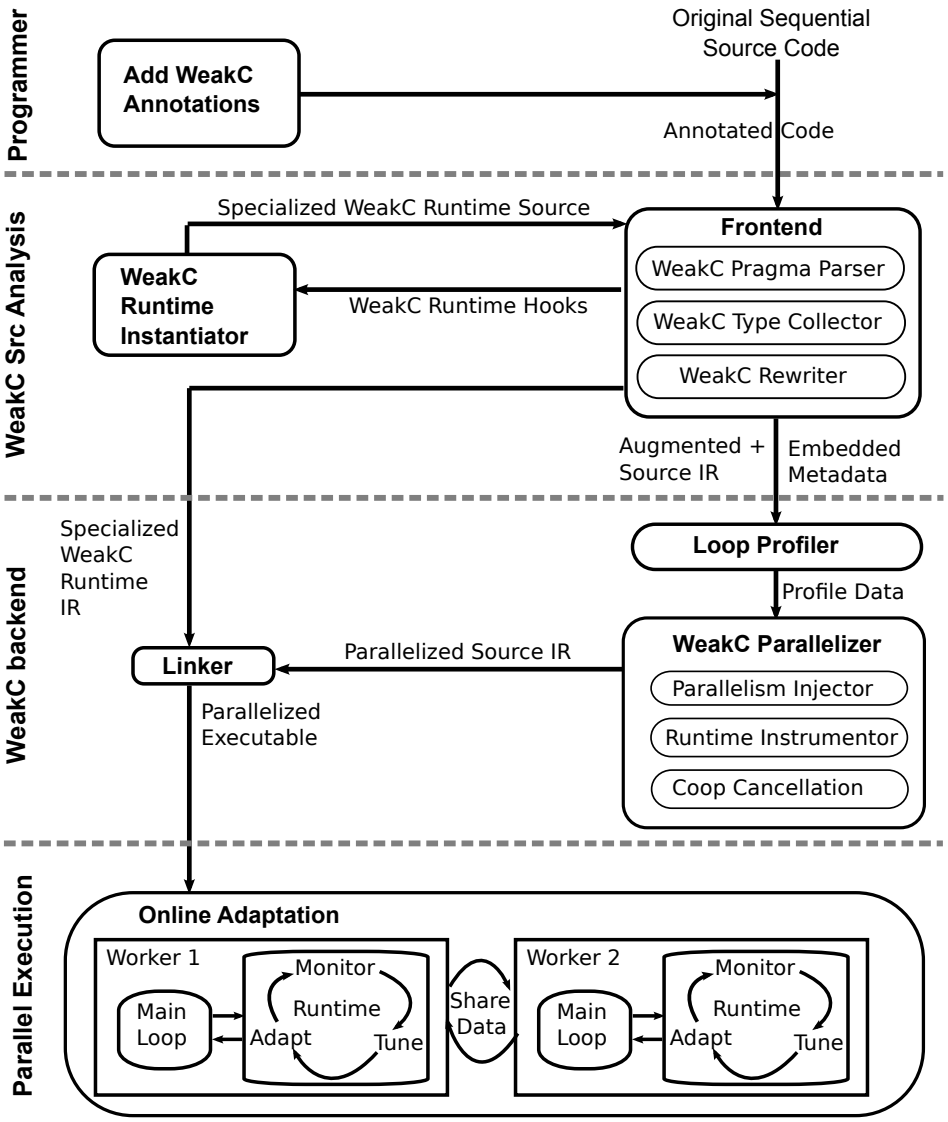


Figure 5.8: The WEAKC Parallelization Workflow

By injecting parallelism in the backend, the WEAKC system can transparently target different parallel execution models. Currently, the WEAKC system uses a `pthreads`-based execution model for shared memory systems, but can be easily targeted for clusters using a MPI-based parallelizer.

The runtime instrumentor augments the search loop with calls into the runtime for sparse sharing. These include calls to save a parallel worker's native weakly consistent set for exchange with other workers, and calls to exchange weakly consistent data between

workers at periods determined by the runtime. The final step in parallelizing the loop handles graceful termination — ensuring cancellation of remaining workers when one worker completes execution. WEAKC follows the *cooperative cancellation* model [196] where the compiler inserts cancellation checks at well-defined syntactic points in the code. In our implementation these points correspond to the return sites of the transitively determined callers into the WEAKC runtime. Finally, the parallelized IR is linked with the specialized runtime IR to generate a parallel executable.

5.6 The WEAKC Runtime

This section describes the WEAKC runtime system.

5.6.1 Design Goals

The design of WEAKC runtime API and system has the following goals:

- First, the API design should have a clear separation of concerns between the subcomponents that achieve parallelism injection and management, synchronization, and tuning, with each subcomponent making minimal assumptions about the other subcomponents. Such a design ensures easy retargeting of the API to multiple parallel substrates. For example, in order to target clusters, there should not be any need to change the tuning algorithms or the sharing protocol, and only calls to low level parallel library primitives need be changed.

- In order to allow for standalone use by expert programmers, the interface presented by the runtime should be relatively medium to higher level, and should not make any assumptions about the use of particular data structures (for example, STL or STAPL) within a client.

Function	Description
Parallelization API	
weakC_initRuntime()	Intialize the WEAKC runtime, setting up various defaults. Creates and sets defaults for globals including termination status and various bookkeeping data structures.
template<typename S, template<typename> class W, typename ER, typename A> weakC_cloneParentObject(S*, int (*)(S*), double (S::*)() const)	Clones the parent data object (of type S, first argument) that contains the weakly consistent set (of type W<ER>). Records a pointer to the search's driver function that uses S and a pointer to a member function of S that estimates search progress
weakC_spawnWorkers()	Spawns parallel workers, each given ownership of a uniquely cloned parent object
weakC_joinWorkers()	Waits for termination of spawned workers
weakC_finiRuntime()	Finalizes the WEAKC runtime. Deletes various data structures
Synchronization API	
template<typename S, template<typename> class W, typename ER, typename A> weakC_saveIntoLocalReplica(id_t, W<ER>&, A&)	Phase I of sparse sharing. Saves weak data elements from a worker's native set into it's local replica within the runtime. Takes in the identity of the worker, a reference to the native set (W<ER>), and a reference to an allocator for ER.
template<typename S, template<typename> class W, into typename ER, typename A> weakC_copyFromRemoteReplicas(id_t, W<ER>&, A&)	Phase II of sparse sharing. Pulls in weakly consistent data from remote worker's local replica into one's own native version, which is passed in as the second argument. Copying is done with a distributed and point to point synchronization mechanism.
Tuning API	
weakC_initOptimizer()	Initializes tuning by binding tuning algorithm state to each parallel worker
weakC_registerTunableParam(id_t, int*, int lb, int ub)	Register a tunable parameter from a worker, along with is upper and lower bounds.
weakC_tune(id_t)	Tune the parallel configuration of worker identified by id_t
weakC_deregisterTunableParam(id_t, int*)	Deregister a tuning parameter for worker identified by id_t
weakC_finiOptimizer()	Finalize tuning algorithm state, and detach from parallel workers

Table 5.3: The WEAKC Runtime API

In order to achieve the above goals, the WEAKC runtime maintains a clear separation between its parallelization, synchronization, and tuning subsystems. Although its current implementation is POSIX-threads based, no interface changes are required to port to parallel runtimes like Intel's TBB or OpenMP. Furthermore, the runtime interface is based on generic data types that can be targeted by compilers and programmers alike and there is no dependence on libraries like C++ STL. Table 5.3 shows the runtime API and Figure 5.9 shows a schematic of WEAKC's parallel runtime execution.

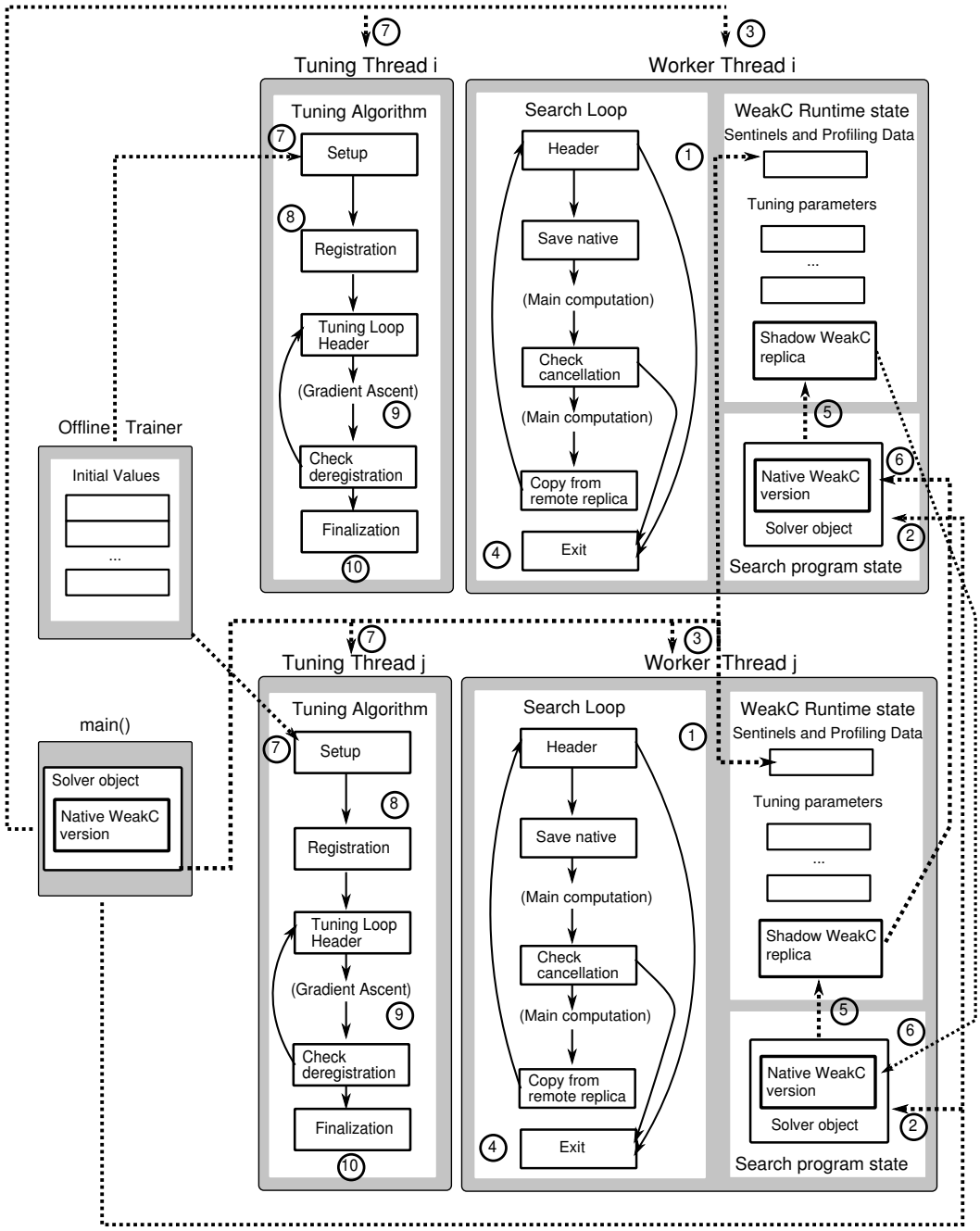


Figure 5.9: The WEAKC runtime execution model

5.6.2 The Parallelization subsystem

This component controls parallelism injection, management, and termination as follows:

- **Initialization** (1): Records initial values of tuning parameters from the environment, prepares sentinels for starting and stopping worker and tuning threads; initializes bookkeeping and profiling data.

- **Replicating Search Objects** (2): Following an object-based concurrency model [114], the WEAKC runtime maintains a one-to-one map between parallel workers and replicated search objects. The replication function uses a reference to the original solver object declared in the sequential program to create new solver objects that are recorded as runtime state. Additionally, functors encapsulating (a) the main driver that performs the search and (b) a progress measure function for online adaptation are recorded internally in the runtime.

- **Spawning Parallel Workers** (3): Creates independent threads of control, each given ownership of its unique search object, initialized to search from a distinct point in the search space.

  The current implementation is based on the POSIX threading library. Once created, each thread invokes a functor that encapsulates the driver search function (recorded earlier) to start its search.

- **Finalization** (4): Upon completing its execution successfully, a worker sets a global cancellation sentinel flag which is polled by all workers at appropriate cancellation points. The other workers thus quickly exit their search and terminate gracefully. The finalization routine then releases internal resources associated with each worker such as replicated memory for weak data structures, destroys locks and barriers, closes profile data files and returns the identity of the successful solver along with its return status for continued use within the original sequential program.

5.6.3 The Synchronization subsystem

The synchronization subsystem orchestrates sparse sharing in two phases: in first phase, data from a worker's *native* weakly consistent data structure is copied into a *shadow replica* that resides within the WEAKC runtime. In the second phase, remote workers pull data from other worker's shadow replica into their own native version.

Sparse sharing was designed according to this two-phased protocol for several reasons. First, a shadow replica distinct from a native version allows remote workers to exchange data asynchronously with a more efficient coarser grained synchronization mechanism. The data exchange is decoupled from a local worker's operations on its native version, thereby preventing incorrect program behavior due to data races. Second, two-phased sharing has the effect of buffering: given a sequence of insert and delete operations on a native version, only its net result is carried over into the shadow replica, effectively amortizing the synchronization costs over this sequence. Third, this protocol naturally leads to a sharing discipline that avoids deadlocks. In particular, no remote worker ever requests access to a worker's shadow replica while holding access to another worker's shadow replica and so hold-and-wait conditions necessary for a deadlock do not arise.

Operationally, synchronization is done as follows:

1. **Saving into shadow replica** (5): Given a native version of a weakly consistent set, iterates through the elements of the set adding them to the shadow replica. If the shadow replica is full, an existing element is chosen and replaced using a comparison function designated by WEAKC annotations. All operations on the shadow replica are performed within a critical section to prevent remote workers from accessing the replica's intermediate state.

2. **Copying from remote shadow replica** (6): Copies data from a remote worker's shadow replica into its own native version. This function is invoked asynchronously

by each worker. Interference may arise only when two or more workers attempt to copy data from the same shadow replica. Each worker attempts to copy data by acquiring exclusive access to a shadow replica in sequence, with the order of acquisition and release determined apriori. As above, existing elements in a native version are replaced if needed using a comparison function.

The API for a weakly consistent set is shown in Table 5.3, while that for a map takes in a key, value pair instead of a single element and is omitted for brevity. The synchronization interface includes an optional allocator object to support custom memory management of weakly consistent data structures. In such a case, an instance of this allocator class is used within the WEAKC runtime to manage a shadow replica's memory allocation and deallocation.

5.6.4 The Tuning subsystem

The tuning component implements WEAKC's online adaptation and allows for both synchronous [197] and asynchronous models of tuning [199]. Synchronous tuning is done as part of a worker's thread of computation, while asynchronous tuning is done concurrently by tuner threads that update tunable program state within a worker periodically. The asynchronous model has minimal impact on original search computation since the cost of the tuning algorithm itself is not borne by the search process. At the same time, tuning of variables within search process's state can be done transparently by the tuner without affecting the rest of search program state. It is also possible to plug in a variety of tuning algorithms without affecting rest of the library. The main advantage of synchronous tuning is that the effect of tuning is immediately visible on search program's computation and no

additional tuning threads need to be created. The WEAKC library maintains a clear separation between tuning algorithm state and the search/parallel program state. Tuning works as follows:

- **Setup** (7): Initializes tuning algorithm state, including seed values for tuning parameters from the environment. For asynchronous tuning, lightweight tuning threads are created as part of the setup – one for each parallel worker.

- **Registration of tunable parameters** (8): Records pointers to tuning variables along with lower and upper bounds for these variables to constrain tuning to within these limits. For example, by constraining set size to not be negative, tuning algorithms are prevented from necessarily exploring infeasible regions in a search space.

- **Tuning** (9): Tuning of parameters for each solver is done independently of each other. Each tuner perturbs variables registered for tuning, observes the effect of perturbation by estimating change in progress using the progress measure functor recorded internally within WEAKC, and then uses it to perform optimization.

- **Deregistration and Finalization** (10): Parameters can be de-registered on the fly to disable tuning during certain phases of computation. A tuning algorithm terminates when a worker completes its search. In case of asynchronous tuning, pre-emptive thread cancellation primitives are employed to finalize tuning threads. Cancellation routines are invoked instead of the cooperative cancellation used for graceful termination of worker threads because cancellation primitives cause tuner threads to wake from sleep immediately (`usleep` is a POSIX thread cancellation point). Tuning threads sleep for the majority of their execution waiting to measure the results of their perturbation on worker state, so cancellation routines save valuable time. This is especially beneficial when WEAKC is entered multiple times with the main program (as in qmaxsat, see Section 6.2).

5.6.5 Online adaptation

Desired Properties. Online adaptation within WEAKC poses a number interesting challenges. First, concurrent access to a search program's runtime state by both a worker and a tuning thread (in asynchronous tuning) during execution can result in inconsistencies if mutual exclusion is not ensured, or in performance penalties otherwise. In WEAKC, parameters relating to shadow replicas within the runtime library are tuned, which a search thread has no access to, except during phases of synchronization that are explicitly controlled by the runtime. Second, online adaptation is useful only when programs run sufficiently long for a tuner to sample enough parameter configurations to optimize at runtime. Search and optimization programs typically satisfy this requirement due to the intractability of the underlying problems. Third, tuning algorithms should be lightweight, and should not interfere with and slow down the main search algorithm computation. This implies that a tuning algorithm should either be invoked infrequently or involve only relatively inexpensive computations, and should not increase contention for hardware resources shared with the search threads. The tuning algorithm used within WEAKC has these properties.

Approach. The goal of online adaptation in WEAKC is to improve the overall execution time of parallel search. Online tuning systems for programs having parallel code sections with regular memory accesses that are invoked multiple times can monitor the execution time of those code sections in response to changes in tuning parameters and recalibrate accordingly. However, search programs typically contain one main loop that is iterated many times and do not exhibit regular memory access patterns. Hence, we rely on application level progress measures that serve as a proxy for online performance. Most solvers already provide such a measure that is conveyed to WEAKC using an annotation.

WEAKC tuning strives to maximize progress of each worker by tuning parameters that are implicitly exposed by the weakened consistency semantics of the auxiliary data structures. The current implementation tunes three parameters: (a) the replica save period that

determines how often elements are saved from native to shadow replica (b) the replica exchange period that determines how often elements are copied from remote worker's shadow replica to native version and (c) sharing set size – the number of elements that are saved and copied between parallel workers. The WEAKC tuning algorithm works by perturbing the values of these parameters, waiting for the impact of these changes to reflect in sharing, measuring the difference in progress effected by tuning and incorporating this feedback for online adaptation. The WEAKC runtime currently includes an online tuning algorithm based on gradient ascent, implemented under an asynchronous model of tuning. In this model, tuning is done concurrently in a separate thread and has minimal interference on the main thread's computation.

Algorithm 3: *Offline selection of parameter values*

for $inp \in \mathcal{I}_1$ **do**
  $env \leftarrow (nthreads = 1)$;
  $(n_{iter}, \mathcal{D}_{size}) \leftarrow prog(env, inp)$;
  $size_{am} = (\sum_{i=1}^{|\mathcal{D}_{size}|} \mathcal{D}_{size}(i))/\,|\mathcal{D}_{size}|$;
  $(gm_{size}, gm_{iter}) \leftarrow (gm_{size} * size_{am}, gm_{iter} * n_{iter})$;
end
$(gm_{size}, gm_{iter}) \leftarrow (gm_{size}^{1/|\mathcal{I}_1|}, gm_{iter}^{1/|\mathcal{I}_1|})$;
for $i = 1$ *to* $\lfloor log(gm_{iter}) \rfloor$ **do**
  $(n_{xchg}, n_{save}) \leftarrow (10^i, 2 * 10^i)$;
  $\mathcal{D}_{sharing} \leftarrow \mathcal{D}_{sharing} \bullet (n_{xchg}, n_{save})$;
end
for $i = 1$ *to* $|k|$ **do**
  $\mathcal{D}_{size} = \mathcal{D}_{size} \bullet (gm_{size}/i) \bullet (gm_{size} * i)$;
end
for $inp \in \mathcal{I}_2$ **do**
  for $s \in \mathcal{D}_{size} \wedge (n, m) \in \mathcal{D}_{sharing}$ **do**
    $env \leftarrow (size = s, xchg = n, save = m)$;
    $\mathcal{X}(inp, s, n, m) \leftarrow \mathcal{T}(prog_{par}(env, inp))/\mathcal{T}(prog_{seq}(env, inp))$;
  end
  $\mathcal{S}(s, n, m) \leftarrow \mathcal{S}(s, n, m) * \mathcal{X}(inp, s, n, m)$;
end
for $s \in \mathcal{D}_{sharing} \wedge (n, m) \in \mathcal{D}_{size}$ **do**
  $\mathcal{S}(s, n, m) \leftarrow \mathcal{S}(s, n, m)^{1/|\mathcal{I}_2|}$;
end
$(s_{ini}, xchg_{ini}, save_{ini}) \leftarrow \arg\max_{(s,n,m) \in \mathcal{D}_{size} \times \mathcal{D}_{sharing}} \mathcal{S}(s, n, m)$;

Profile-based offline selection of initial parameter values. The initial values of parameters for tuning are determined based on profiling. First, approximate measures for typical WEAKC set sizes and number of iterations for convergence are determined. This

is done by sampling the *sequential* runs of a set of randomly selected inputs that run for at least one minute. Based on these measures, a range of potential starting values for each parameter is selected for offline training. The combined space of parameter values is prohibitively large, hence the selected range is obtained by sampling at regular periods around the approximate measures computed earlier. Next, offline training is done using a second input set by invoking a *parallelized* version of the search program that performs sparse sharing. In this program, values for each parameter (like sharing period, set size) are statically set at the beginning of the program and do not change during execution. This parallel version is invoked for every combination of parameter values computed in the prior step, and corresponding speedup is measured. The particular combination of parameter values that results in the best geometric mean speedup across all inputs is then selected as the initial condition to guide online adaptation. This combination also constitutes the setting for the non-adaptive version of WEAKC evaluated in Section 6.2. Algorithm 3 gives the outline of the profile-based offline parameter selection algorithm.

Algorithm 4: *Online adaptation using gradient ascent*

```
1  def gradient(id, x̃) :
2      a⃗ ← x⃗
3      for i = 1 to dim(a⃗) do
4          a⃗_i ← x⃗_i + δ_i
5          f_1 ← perturbAndMeasure(id, a⃗)
6          a⃗_i ← x⃗_i − δ_i
7          f_2 ← perturbAndMeasure(id, a⃗)
8          ▽_i = (f_2 − f_1)/(2 × δ_i)
9      end
10 x⃗ ← x⃗_ini  where x⃗_ini  is from offline parameter selection
11 for i = 1 to NUMITERS do
12     ▽f ← gradient(id, x⃗)
13     if (‖▽f‖ < ϵ) then
14         return x⃗
15     end
16     x⃗ ← x⃗ + α ∗ ▽f
17 end
18 return x⃗
```

Online adaptation using gradient ascent. The online adaptation algorithm used within WEAKC is based on gradient ascent (Algorithm 4). It is invoked independently for each

worker, with the goal of determining the parameter configuration that maximizes the progress made by each worker. The parameters correspond to those exposed by the WEAKC data structures, and the objective function is regarded as a function of these parameters. Starting with the initial values seeded by the offline profiling based parameter selection algorithm, the tuning algorithm first perturbs these values in either direction for each parameter (Lines 4 to 7) and computes the gradient of the objective by measuring the difference in progress made due to this perturbation (Line 8). Because the objective function is not a direct function of the WEAKC parameters, measurement is performed only after sufficient number of sharing cycles have elapsed beyond the perturbation point (within `perturbAndMeasure` on Lines 5 and 7). Once the gradient has been computed, the parameter configuration is updated in the direction of the gradient, scaled appropriately by a fraction α. This whole cycle is iterated until the gradient norm is very small (which would be the case near a local maxima) or for a fixed number of iterations (Lines 15 to 17).

Chapter 6

Experimental Evaluation

This chapter describes the evaluation of the COMMSET and WEAKC programming models. Each model is evaluated on set of programs sourced from a variety of benchmark suites and on case studies of programs selected from the field study.

6.1 Commutative Set evaluation

The COMMSET programming model is evaluated on a set of twenty programs. These programs along with the details of their parallelization is shown in Table 6.1 and Table 6.2. The programs were selected based on the following criteria:

- The number of actual lines of code for each program, excluding comments and whitespaces, should be greater than a reasonable minimum (400 in our experiments). In our experiments, the relevant lines of code were measured using `cloc` [60] tool.
- Programs should be written in an imperative language like C or C++ on which COMMSET pragmas can be applied.
- Programs should have at least 40% of their execution time spent in loops for their representative inputs.

- Overall, the entire collection of candidate programs should represent a diverse set of application domains.

Additionally, the selection generally avoided programs that have been shown to be scalably parallelized by prior automatic parallelization techniques.

Each candidate program is evaluated for both applicability and performance of different parallelization schemes with and without semantic changes. First, a program is attempted to be parallelized with data parallel (DOALL) and pipelined (PS-DSWP) parallelization techniques without the use of any commutativity extensions. If a program is scalably parallelized with DOALL, the DOALL scheme is marked as applicable, its performance noted, and no further parallelization/annotations are applied for this program. If DOALL scheme fails to obtain scalable speedup *due* to its inapplicability to the dependence patterns exhibited in the hot loops, then PS-DSWP parallelization is applied and its performance noted.

Only when neither DOALL or PS-DSWP is inapplicable due to the presence of restrictive loop-carried dependence patterns within the hot loops, the program's source is examined to determine if certain execution orders can be relaxed without changing its intended behavior. If the relaxation only requires the application of basic Commutative [41] annotation to obtain scalable speedup, then COMMSET extensions are not evaluated on the candidate program. Only when a given program needs semantic changes for scalable parallelization beyond the basic Commutative primitive, the COMMSET primitives are applied at all and the resulting performance is measured. Apart from evaluating parallelization schemes enabled by COMMSET primitives in these programs, alternative parallelization schemes without using COMMSET primitives are also evaluated for performance comparison.

All evaluations are carried out on a 1.6GHz Intel Xeon 64-bit dual-socket quad core machine with 8GB RAM that runs Linux 2.6.24. The candidate programs and their resulting parallelization, described in the following subsections are categorized into four buckets:

- Programs parallelizable without semantic changes. The semantic changes are either not needed for scalability or cannot be applied due to strict conformance of these programs to the sequential model.
- Programs parallelizable with basic Commutative.
- Programs parallelizable with Commutative Set.
- Programs not parallelizable with Commutative Set. These programs either do not scale with Commutative Set, or do not exhibit execution orders relaxable by Commutative Set.

6.1.1 Parallelizable without semantic changes

Of the twenty programs, thirteen programs were parallelizable without any semantic changes. Of these, four programs exhibited code patterns whose execution orders could be relaxed further via commutativity assertions. The result of parallelizing the remaining nine programs are shown in Figure 6.1 and Figure 6.2. The rest of this section discusses the parallelization of each program in detail.

`171.swim` is a program that performs shallow water modeling, originally written in Fortran 77 and converted to C using the f2c [77] tool. Although the original Fortran program is array based, the conversion introduces pointers and structures within the C program. The majority of execution time is spent in four loops all of which were parallelized with DOALL. However, each loop is invoked a large number of times and the time spent per invocation inside each loop are relatively small. The resulting parallelism overheads limit the overall speedup to 4.2x.

`myocyte` is a program that simulates heart muscle cell behavior. The sequential version of this program was parallelized with DOALL and scales well till eight threads to give

Program	Origin	Application Domain	Target Function(s)	Exec Time of Target Loops	Total Src LOC	LOC Changed	LOC Change Desc.
171.swim	SPEC2000 [189]	Meteorology	`calc_1`, `calc_2`, `calc_3`, `inital`	99%	1338	0	-
myocyte	Rodinia [53]	Heart modeling	`main`	100%	1815	0	-
stringsearch	MiBench [90]	Office	`main`	100%	2943	3	Privatization
disparity	SD-VBS [207]	Computer Vision	`finalSAD`	76%	2202	0	-
rsearch	MineBench [151]	RNA Sequencing	`serial_search_database`	100%	21624	0	-
EBGMFaceGraph	FacePerf [39]	Graph Matching	`main`	100%	10428	0	-
facerec	ALPBench [133]	Face Recognition	`readAnd-Project-Image`	96%	11106	0	-
ecbdes	VersaBench [169]	Encryption	`driver`	100%	1302	2	Privatization
autocorr	EEMBC [55]	Telecomm	`t_run_test`	100%	681	2	Privatization
url	NetBench [144]	Network Processing	`main`	100%	629	1	Privatization
crc	NetBench [144]	Error Correction	`main`	100%	10284	0	-
md5sum	Open Src [20]	Security	`main`	100%	470	1	Privatization
456.hmmer	SPEC2006 [97]	Comp. Biology	`main_loop_serial`	99%	20658	2	Privatization
geti	MineBench [151]	Data Mining	`FindSomeETIs`	98%	889	2	Privatization
ECLAT	MineBench [151]	Data Mining	`newApriori`	97%	3271	0	-
potrace	Open Src [184]	Graphics	`main`	100%	8292	25	Privatization
kmeans	STAMP [147]	Data Clustering	`work`	99%	516	0	-
em3d	Olden [45]	Material Simulation	`initialize_graph`	97%	464	0	-
tracking	SD-VBS [207]	Computer Vision	`main`	99%	2509	0	-
ga	HPEC [93]	Search	`main`	99%	811	0	-

Table 6.1: Sequential programs evaluated, their origin and domain, functions enclosing the target loops, execution time spent in these loops, lines of source code as measured by `cloc` [60], sequential non-semantic changes made before parallelization

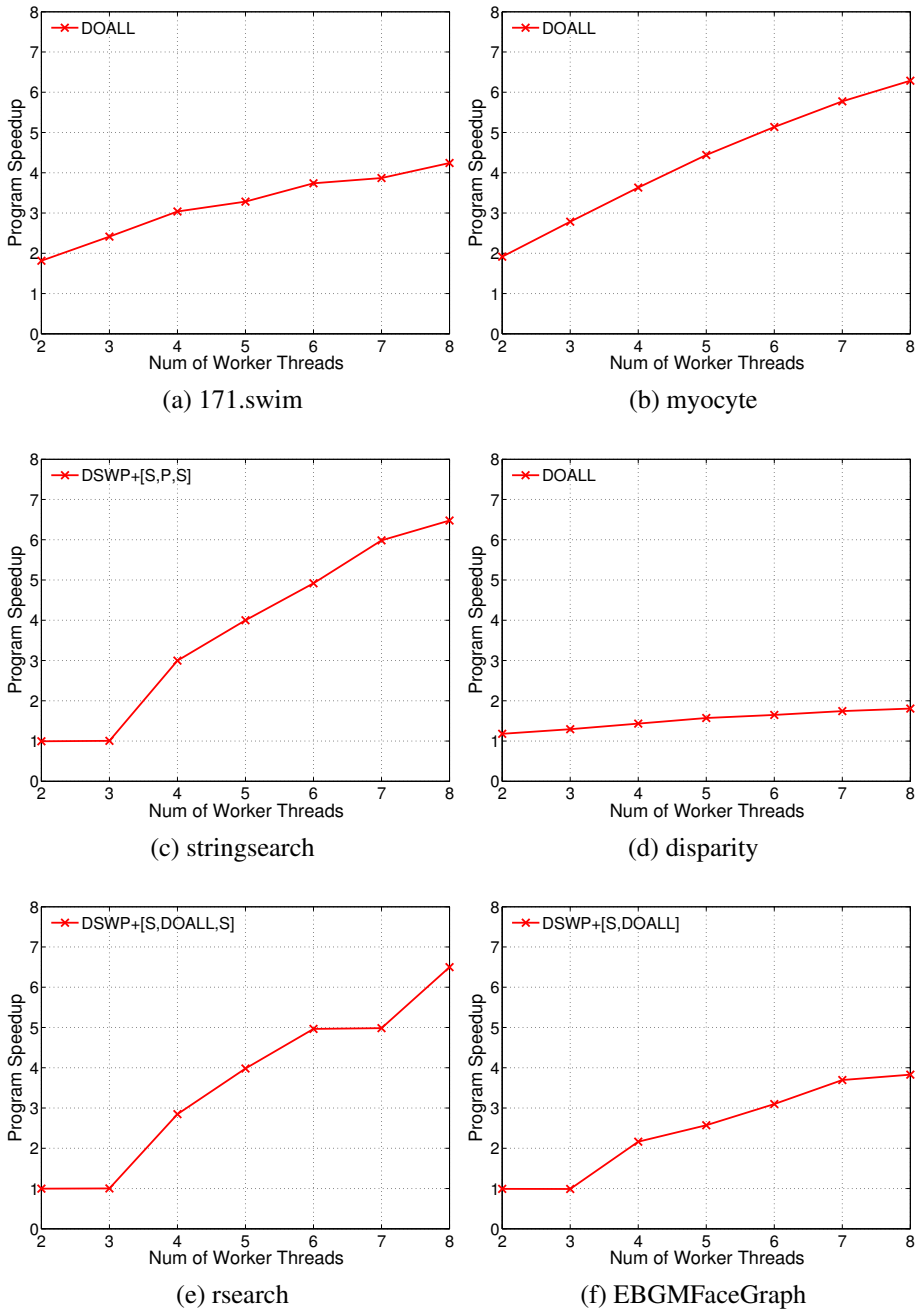
a speedup of around 6.3x. `stringsearch` is a program that finds a set of search strings within a corpus of text, and prints out whether the search string was found or not. Since the prints to the console do not specify the input string being searched for, the prints cannot be

Program	Num of COMMSET Annotations	COMMSET attributes	Applicable Parallelizing Transforms	Best Speedup	Best Parallelization Scheme	Num of custom AA rules
171.swim	0	-	DOALL	4.2x	DOALL	4
myocyte	0	-	DOALL	6.3x	DOALL	1
stringsearch	0	-	PSDSWP	6.5x	PS-DSWP	4
disparity	0	-	DOALL	1.8x	DOALL	0
rsearch	0	-	PS-DSWP	6.5x	PS-DSWP	5
EBGMFaceGraph	0	-	PS-DSWP	3.8x	PS-DSWP	2
facerec	0	-	PS-DSWP	1.9x	PS-DSWP	2
ecbdes	0	-	DOALL	7.0x	DOALL	0
autocorr	0	-	DOALL	7.6x	DOALL	0
crc	1	I, S	DOALL, PS-DSWP	3.3x	DOALL + Spin	2
url	2	I, S	DOALL, PS-DSWP	7.7x	DOALL + Spin	6
md5sum	10	PC, C, S&G, O	DOALL, PS-DSWP	7.6x	DOALL + Lib	0
456.hmmer	9	PC, C&I, S&G, O	DOALL, PS-DSWP	5.8x	DOALL + Spin	18
geti	11	PI&PC, C&I, S&G	DOALL, PS-DSWP	3.6x	PS-DSWP + Lib	23
ECLAT	11	PC, C&I, S&G	DOALL, PS-DSWP	7.5x	DOALL + Mutex	28
potrace	10	PC, C, S&G	DOALL, PS-DSWP	5.5x	DOALL + Lib	0
kmeans	1	C, S	DOALL, PS-DSWP	5.2x	PS-DSWP	4
em3d	8	I, S&G	DSWP, PS-DSWP	5.8x	PS-DSWP + Lib	7
tracking	10	PC, C&I, S&G, O	DOALL	3.3x	DOALL + Spin	13
ga	5	PC, C, S&G	OR parallelism	0.94x	OR + Spin	0

Table 6.2: Sequential Programs evaluated, number of lines of commutativity annotations added, the various COMMSET attributes used (PI: Predication at Interface, PC: Predication at Client, C: Commuting Blocks, I: Interface Commutativity, S: Self Commutativity, G: Group Commutativity, O: Optional Commuting Block), applicable parallelizing transforms, the best speedup obtained and the corresponding (best) parallelization scheme, and the number of custom alias disambiguation rules employed

reordered and so a PS-DSWP schedule was the result of parallelizing the main loop. This parallelization gets the benefit of buffering prints in the last stage to obtain a speedup of 6.5x on eight threads.

`disparity` is a computer vision program that computes the depth information based on objects represented in two pictures. The candidate loop for parallelization does array computation on different sections of an image and is DOALLable. The resulting speedup is limited to 1.8x on eight threads primarily due to only 76% of execution time spent inside



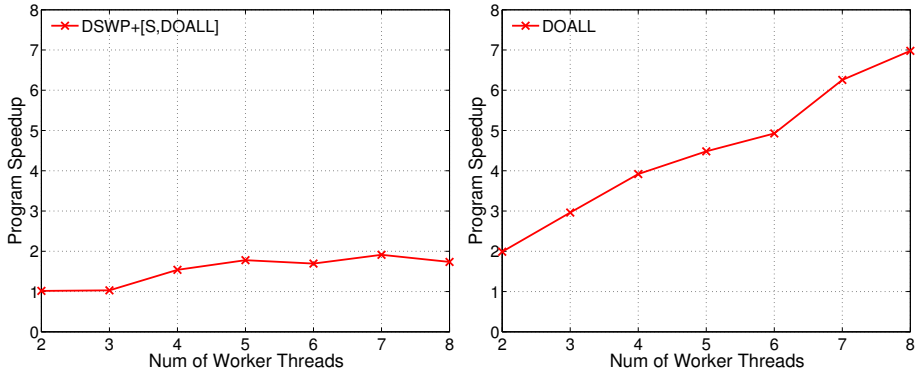
(a) 171.swim (b) myocyte

(c) stringsearch (d) disparity

(e) rsearch (f) EBGMFaceGraph

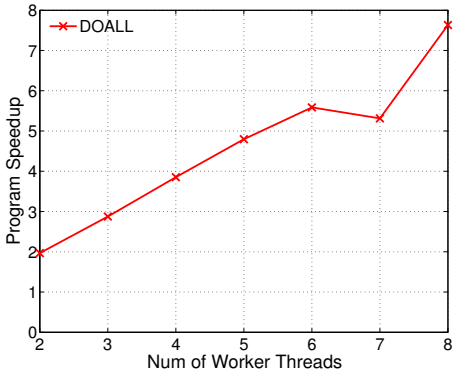
Figure 6.1: Performance of the best parallelization schemes on applications that were parallelizable without any semantic changes (first six out of nine programs).

the loop and its high invocation count. `rsearch` is a program that searches gene databases



(a) facerec

(b) ecbdes



(c) autocorr

Figure 6.2: Performance of the best parallelization schemes on applications that were parallelizable without any semantic changes (last three out of nine programs).

for RNA sequences. The sequential version of this program was parallelized using a three-stage PS-DSWP pipeline, with the first stage reading search sequences, the parallel stage performing the search in the database, and the final stage writing the result of the search to its output. A peak speedup of 6.5x was obtained on eight threads.

`EBGMFaceGraph` is a program within a repository of implementations of face recognition algorithms that read a sequence of images and constructs "face graphs" for each image. This graph building step is implemented in the form of a doubly nested loop of linked list traversals of which the innermost loop was parallelized using PS-DSWP to yield

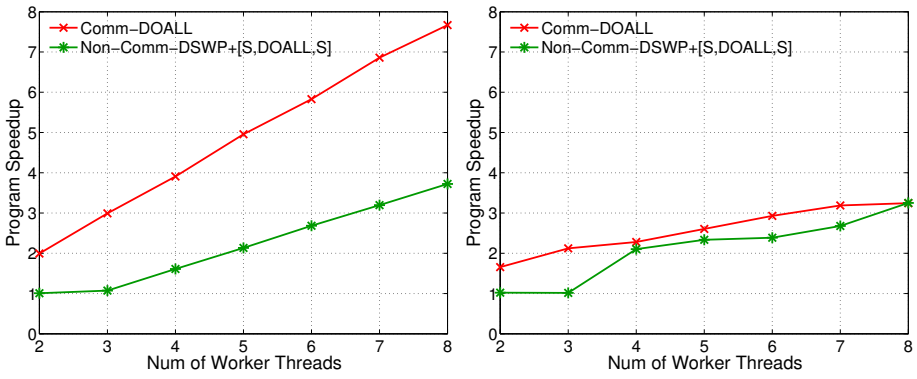
a speedup of 3.8x on eight threads. `facerec` is a program for face recognition that performs a projection of each image on a subspace matrix in one of its phases. This projection step is again implemented as a doubly nested loop of linked list traversals to which PS-DSWP was applied to get a speedup of 1.9x. The limited speedup is a result of spending only 96% of execution time in the main loop and overheads associated with parallel execution.

`ecbdes` is a program that does encryption of textual data using the Data Encryption Standard (DES) [57] algorithm. The main loop in this program performs encryption of input data for a number of times repeatedly. This loop was parallelized using DOALL and the parallelization scales well to yield a speedup of 7x on eight cores. `autocorr` is a program that perform auto-correlation of input data for a number of iterations. The parallelization of this auto-correlation loop yields a speedup of 7.6x on eight threads.

6.1.2 Parallelizable with basic Commutative

URL: url based switching

The main loop in the program switches a set of incoming packets based on its URL and logs some of the packet's fields into a file. The underlying protocol semantics allows out-of-order packet switching. Marking the function to dequeue a packet from the packet pool and the logging function as self commutative broke all the loop carried flow dependences. No synchronization was necessary for the logging function while locks were automatically inserted to synchronize multiple calls to the packet dequeuing function. A two stage PS-DSWP pipeline was also formed by ignoring the commutativity annotation on the packet dequeue function. The DOALL parallelization (7.7x speedup on eight threads) outperforms the PS-DSWP version (3.7x on eight threads) because of low lock contention on the dequeue function and the overlapped parallel execution of the packet matching computation.



(a) url (b) crc

Figure 6.3: Performance of DOALL and PS-DSWP schemes using the basic Commutative extension. Parallelization schemes in each graph's legend are sorted in decreasing order of speedup on eight threads, from top to bottom. The DSWP + [...] notation indicates the DSWP technique with stage details within [...] (where *S* denotes a sequential stage and *DOALL* denotes a parallel stage). Schemes with *Comm-* prefix were enabled only by the use of basic Commutative extension. For each program, the best Non-Commutative parallelization scheme, obtained by ignoring the basic Commutative extension is also shown.

crc: Checksums for Internet packets

`crc` is a program that successively dequeues incoming packets from an internet queue, computes a cyclic redundancy checksum on each packet and accumulates these checksums. Since the dequeuing of the packets can be done out-of-order without changing the semantics of the program, the dequeue function was marked as self-commutative at the interface level. With this relaxation, the COMMSET compiler is able to DOALL this loop to obtain a speedup of 3.2x on eight threads, the speedup limited due to locking overheads in the high-frequency dequeue operation. As an alternate parallelization, without using commutativity annotation, the main loop in this program can be parallelized using a three-stage PS-DSWP pipeline, with the first stage dequeuing packets, the checksum computed in a parallel stage and the final sequential stage accumulating the checksum. This parallelization also yields a speedup of 3.2x on eight threads.

6.1.3 Parallelizable with Commutative Set

456.hmmer: Biological Sequence Analysis

456.hmmer performs biosequence analysis using Hidden Markov Models. Every iteration of the main loop generates a new protein sequence via calls to a random number generator (RNG). It then computes a score for the sequence using a dynamically allocated matrix data structure, which is used to update a histogram structure. Finally, the matrix is deallocated at the end of the iteration. By applying COMMSET annotations at three sites, all loop carried dependences were broken: (a) The RNG was added to a SELF COMMSET since any permutation of a random number sequence still preserves the properties of the distribution. (b) The histogram update operation was also marked self commuting, as it performs an abstract SUM operation even though the low-level statements involve floating point additions and subtractions. (c) The matrix allocation and deallocation functions were marked as commuting with themselves on separate iterations. Overall, the DOALL parallelization using spin locks performs best for eight threads, with a program speedup of about 5.82x. A spin lock works better than mutex since it does not suffer from sleep/wakeup overheads in the midst of highly contended operations on the RNG seed variable. The three stage PS-DSWP pipeline, gives a speedup of 5.3x (doing better than the mutex and TM versions of DOALL) by moving the RNG to a sequential stage, off the critical path.

GETI: Greedy Error Tolerant Itemsets

GETI is a C++ data mining program that determines a set of frequent items that are bought together frequently in customer transactions (*itemsets*). *Itemsets* are implemented as `Bitmap` objects, with items acting as keys. Items are queried and inserted into the `Bitmap` by calls to `SetBit()` and `GetBit()`. Each *itemset* is inserted into an STL vector and then printed to the console. By adding COMMSET annotations at three sites,

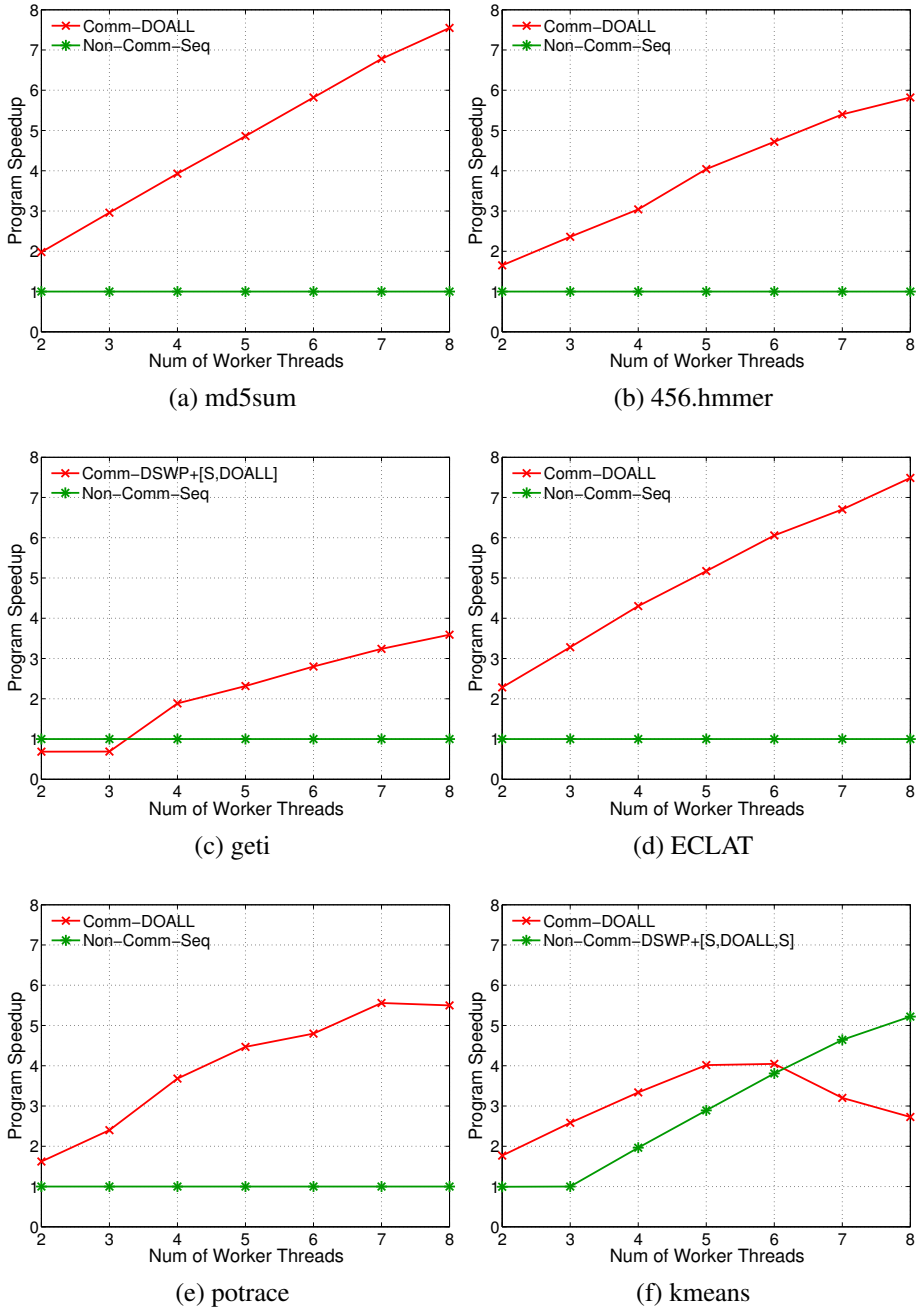
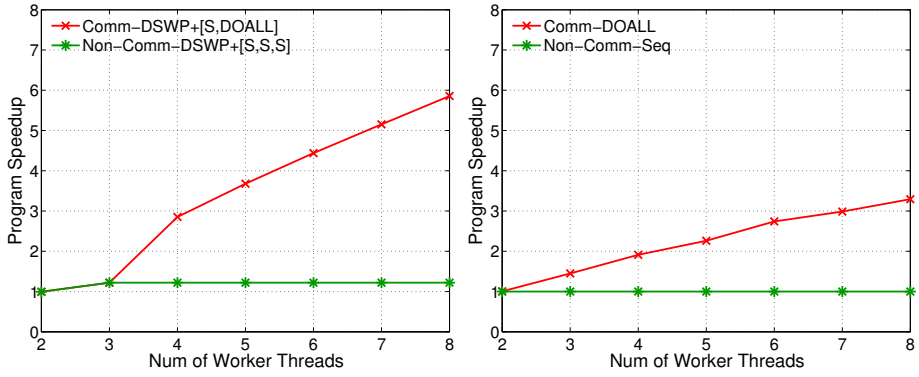
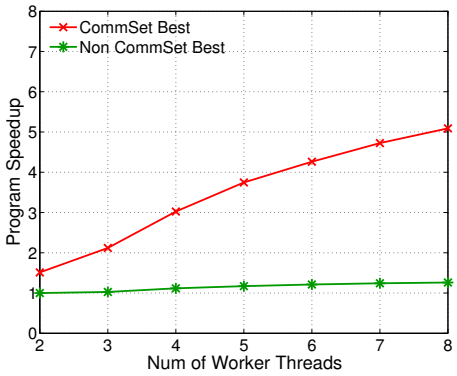


Figure 6.4: Performance of DOALL and PS-DSWP schemes using COMMSET extensions (first six of eight programs). Parallelization schemes in each graph's legend are sorted in decreasing order of speedup on eight threads, from top to bottom. The DSWP + [...] notation indicates the DSWP technique with stage details within [...] (where *S* denotes a sequential stage and *DOALL* denotes a parallel stage). Schemes with *Comm-* prefix were enabled only by the use of COMMSET. For each program, the best Non-COMMSET parallelization scheme, obtained by ignoring the COMMSET extensions is also shown. In some cases, this was sequential execution.



(a) em3d (b) tracking



(c) Geomean over eight programs

Figure 6.5: Performance of DOALL and PS-DSWP schemes using COMMSET extensions (last two of eight programs). Parallelization schemes in each graph's legend are sorted in decreasing order of speedup on eight threads, from top to bottom. The DSWP + [...] notation indicates the DSWP technique with stage details within [...] (where *S* denotes a sequential stage and *DOALL* denotes a parallel stage). Schemes with *Comm-* prefix were enabled only by the use of COMMSET. For each program, the best Non-COMMSET parallelization scheme, obtained by ignoring the COMMSET extensions is also shown. In some cases, this was sequential execution. The last graph compares the geomean speedup of the eight programs parallelized with COMMSET to the geomean speedup of the non-COMMSET parallelization.

the main loop was completely parallelizable with DOALL and PS-DSWP: (a) Itemset constructors and destructors are added to a COMMSET and allowed to commute on separate iterations. (b) `SetBit()` and `GetBit()` interfaces were put in a COMMSET predicated on the input key values, to allow for insertions of multiple items to occur out of order. (c) The code block with `vector::push_back()` and prints was context sensitively marked

as self commutative in client code. The correctness of this application follows from the set semantics associated with the output. The inter-iteration commutativity properties for constructor/destructor pairs enabled a well performing three-stage PS-DSWP schedule. Transactions were not applicable due to use of external libraries and I/O. Although DOALL schemes initially did better than PS-DSWP, the effects of buffering output indirectly via lock-free queues for PS-DSWP and the increasing number of acquire/release operations for DOALL led to a better performing schedule for PS-DSWP on eight threads. PS-DSWP achieved a limited speedup of 3.6x due to the sequential time taken for console prints but maintained deterministic behavior of the program.

ECLAT: Association Rule Mining

ECLAT is a C++ program that computes a list of frequent itemsets using a vertical database. The main loop updates objects of two classes `Itemset` and `Lists<Itemset*>`. Both are internally implemented as lists, the former as a client defined class, and the latter as an instantiation of a generic class. Insertions into `Itemset` have to preserve the sequential order, since the `Itemset` intersection code depends on a deterministic prefix. Insertions into the `Lists<Itemset*>` can be done out of order, due to set semantics attached with the output. COMMSET extensions were applied at four sites: (a) Database read calls (that mutate shared file descriptors internally) were marked as self commutative. (b) Insertions into `Lists<Itemset*>` are context-sensitively tagged as self commuting inside the loop. Note that it would be incorrect to tag `Itemset` insertions as self-commuting as it would break the intersection code. (c) Object construction and destruction operations were marked as commuting on separate iterations. (d) Methods belonging to `Stats` class that computes statistics were added to a unpredicated Group COMMSET . A speedup of 7.4x with DOALL was obtained, despite pessimistic synchronization, due to a larger fraction of time spent in the computation outside critical sections. Transactions are not

applicable due to use of I/O operations. The PS-DSWP transform, using all the COMMSET properties generates a schedule (not shown) similar to DOALL. The next best schedule is from DSWP, that does not leverage COMMSET properties on database read. The resulting DAG-SCC has a single SCC corresponding to the entire inner `for` loop, preventing stage replication.

potrace: Bitmap tracing

potrace vectorizes a set of bitmaps into smooth, scalable images. The code pattern is similar to md5sum, with an additional option of writing multiple output images into a single file. In the code section with the option enabled, the SELF COMMSET annotation was omitted on file output calls to ensure sequential output semantics. The DOALL parallelization yielded a speedup of 5.5x, peaking at 7 threads, after which I/O costs dominate the runtime. For the PS-DSWP parallelization, the sequentiality of image writes limited speedup to 2.2x on eight threads.

kmeans: K means clustering algorithm

kmeans clusters high dimensional objects into similar featured groups. The main loop computes the nearest cluster center for each object and updates the center's features using the current object. The updates to a cluster center can be re-ordered, with each such order resulting in a different but valid cluster assignment. Adding the code block that performs the update to a SELF COMMSET breaks the only loop carried dependence in the loop. The DOALL scheme with pessimistic synchronization showed promising speedup until five threads (4x), beyond which frequent cache misses due to failed lock/unlock operations resulted in performance degradation. Transactions (not shown) do not help either, with speedup limited to 2.7x on eight threads. The three-stage PS-DSWP scheme was best

performing beyond six threads, showing an almost linear performance increase by executing the cluster update operation in a third sequential stage. It achieved a speedup of 5.2x on eight threads. This highlights the performance gains achieved by moving highly contended dependence cycles onto a sequential stage, an important insight behind the DSWP family of transforms.

em3d: Electro-magnetic Wave propagation

em3d simulates electromagnetic wave propagation using a bipartite graph. The outer loop of the graph construction iterates through a linked list of nodes in a partition, while the inner loop uses a RNG to select a new neighbor for the current node. Allowing the RNG routine to execute out of order enabled PS-DSWP. The program uses a common RNG library, with routines for returning random numbers of different data types, all of which update a shared seed variable. All these routines were added to a common Group COMMSET and also to their own SELF COMMSET. COMMSET specifications to indicate commutativity between the RNG routines required only eight annotations, while specifying pair-wise commutativity would have required 16 annotations. Since the loop does a linked list traversal, DOALL was not applicable. Without commutativity, DSWP extracts a two-stage pipeline at the outer loop level, yielding a speedup of 1.2x. The PS-DSWP scheme enabled by COMMSET directives achieves a speedup of 5.9x on eight threads. A linear speedup was not obtained due to the short execution time of the original instructions in the main loop, which made the overhead of inter-thread communication slightly more pronounced.

tracking: Feature Tracking

This is a computer vision program that implements feature extraction as part of gathering motion information from a sequence of images. For the given input parameters, the loop

that processes the remaining frames of each image after the first frame constitutes the majority (99%) of the execution time. This loop involves I/O calls to read image data from the file system and to build in-memory structures. These calls are made within a transitive call chain, inside a `readImage` function. Here, the different I/O calls commute with each other when invoked on separate iterations, even though there could be a case where the file pointers while being different refer to the same file. However, sequential ordering needs to be preserved between I/O calls belonging to the same file. Using a combination of optional commuting blocks and predication, a DOALL parallelization scheme is obtained by the COMMSET compiler which achieves a speedup of 3.3x on eight cores. This program is a good illustration of combining optional commuting blocks and predication. In this case, I/O calls at different points within `readImage` function are added to separate named blocks, which are exported at the `readImage` interface and then bound to the same COMMSET at the call site. At the binding point, commutativity of the named blocks is predicated on the induction variable. Group and self commutativity annotations are applied to break sequential dependences on the image blur and feature tracking operations by allowing the use of any updated image from the last few images (and not necessarily the last image). This has the effect of not using the latest values in an iterative computation, albeit at a higher algorithmic level, with out of order execution. Without breaking this dependence, there is a single strongly connected component that thwarts automatic parallelization without annotation support.

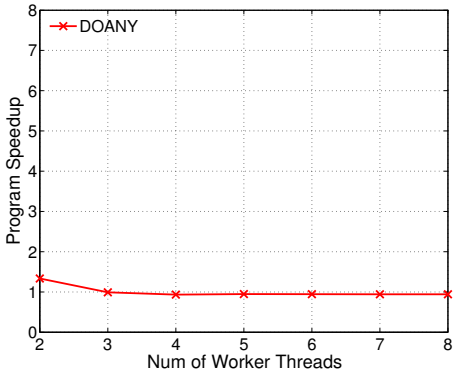
Discussion

The application of COMMSET achieved a geomean speedup of 5.1x on eight threads for eight programs, while the geomean speedup for Non-COMMSET parallelizations on these programs is 1.26x (Figure 6.5(c)). For five out of the eight programs, the main loop was not parallelizable at all without the use of COMMSET primitives. With the application of

COMMSET, DOALL parallelization performs better than PS-DSWP on 6 benchmarks, although PS-DSWP has the advantage of preserving deterministic output in two of them. For two of the remaining programs, PS-DSWP yields better speedup since its sequential last stage performs better than concurrently executing COMMSET blocks in the high lock contention scenarios. DOALL was not applicable for em3d, due to pointer chasing code. In terms of programmer effort, an average of 8 lines of COMMSET annotations were added to each program to enable the various parallelization schemes. Predication based on the client state, as a function of the induction variable enabled well performing parallelizations without the need for runtime checks. The use of commuting blocks avoided the need for major code refactoring. The applicability of COMMSET compared favorably to the applicability of other compiler based techniques like Paralax and VELOCITY. VELOCITY and Paralax cannot be used to parallelize five benchmarks: geti, eclat, md5sum, tracking, and potrace since they do not support predicated commutativity. For 456.hmmer, VELOCITY would require a modification of 45 lines of code (addition of 43 lines and removal of 2 lines) in addition to the commutativity annotations. COMMSET did not require those changes due to the use of named commuting blocks.

6.1.4 Not parallelizable with Commutative Set

`ga` is a program that implements a genetic algorithm for performing graph optimization. The algorithm maintains a population of candidate solutions, represented as chromosomes within the program. It uses operations like mutation, selection and crossover to create new generations of candidate solutions from the current generation and iterates this process to convergence. Within the main iterative loop, a data structure is used to implement elitism where a certain fraction of fittest individuals within certain thresholds are carried forward across generations. The insertion, removal and query operation into this data structure creates loop-carried dependencies.



(a) ga

Figure 6.6: Program that does not scale with COMMSET parallelization.

There are two interesting aspects to the semantics of this loop as far as parallelization is concerned. First, neither DOALL or PS-DSWP can be applied to this iterative loop due to a strict loop-carried data dependence of the form $x = f(x)$ on the computation of a new generation from the previous generations. Instead, a OR style execution model and parallelization can be applied in this case: The original sequential loop is executed in parallel on multiple threads, each on its own replicated memory space, and the entire execution completes if any one thread finishes execution. Second, the operations on the elite chromosome data structure can be marked as commutative, and this relaxation translates into a single shared, concurrent data structure with each operation wrapped as a critical section that is accessed mutually exclusively by the different threads.

However, due to the high frequency of operations on this data structure, the synchronization overheads associated with these operations on a single, concurrent data structure is also very high. As a result, the above parallelization results in no speedup beyond two threads, and in fact slows down the sequential execution by 1.06x on eight threads. However, by applying semantics weaker than commutativity, it is possible to get a much better performance profile for this program (see Section 6.2).

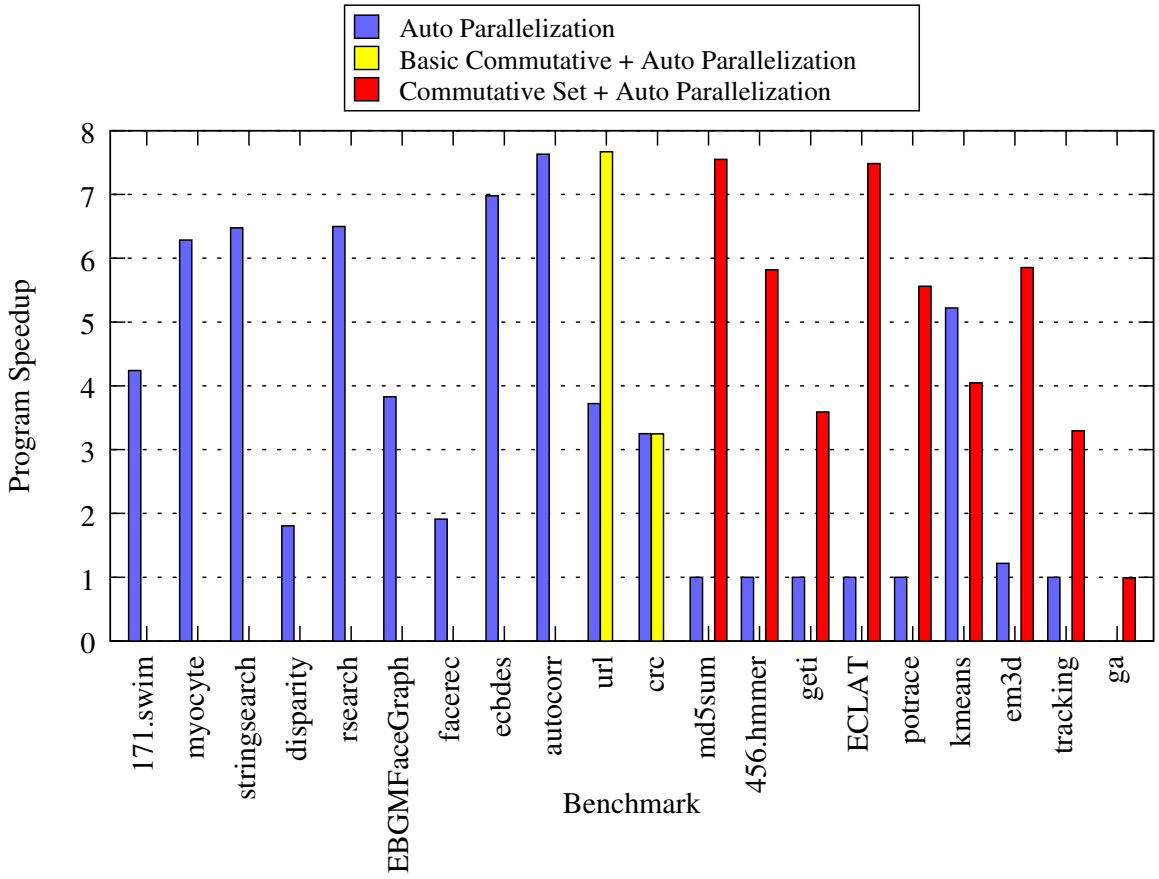


Figure 6.7: Best performance result for each applicable scheme across all 20 program

6.1.5 Summary of results across four categories

Figure 6.7 summarizes the best performance scheme for every applicable scheme in the order in which parallelization was attempted for each of twenty evaluated programs. Figure 6.8 shows the categorization of the results across all the four buckets. Overall, the end-to-end COMMSET parallelization system obtains a speedup of 4x for the twenty programs studied. These twenty program together constitute 102,232 lines of source code. With respect to programming effort, a total of 38 lines of non-semantic code changes were made (primarily for privatization of iteration local data), and a total of 78 commutativity annotations were added. Of these results, automatic parallelization with no commutativity annotations were applicable to 13 programs with a geometric mean speedup of 3.9x over

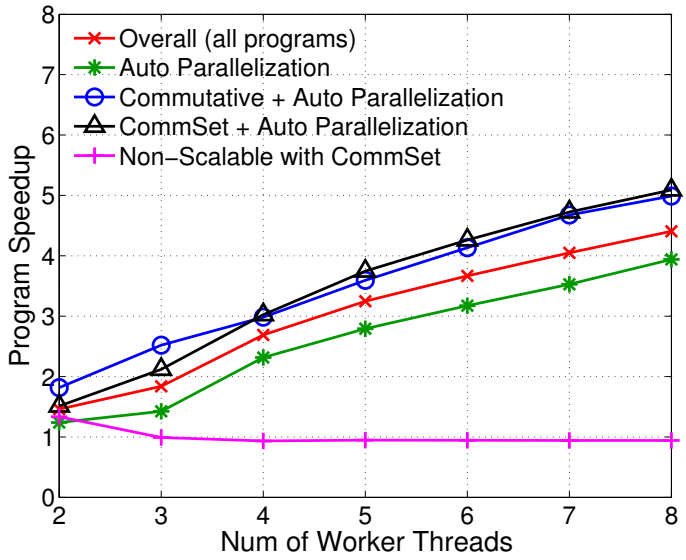


Figure 6.8: Results of parallelization categorized by four buckets (a) Geomean speedup of 13 programs parallelized without semantic changes, marked as `Auto Parallelization` (b) Geomean speedup of 2 programs parallelized with Commutative, marked as `Commutative + Auto Parallelization` (c) Geomean speedup of 8 programs parallelized with COMMSET, marked as `CommSet + Auto Parallelization` (d) Speedup of program that does not scale with COMMSET, marked as `Non-Scalable with CommSet` (e) Overall geomean speedup of all all 20 programs

these programs. With the basic Commutative annotation, the applicability extended to two more programs with geometric mean speedup of 5x. The generalized commutativity annotations provided by the COMMSET language increased applicability to eight programs yielding a geometric speedup of 5.1x.

6.2 WEAKC Evaluation

WEAKC is evaluated on five open source sequential search/optimization programs shown in Table 6.3. They were selected based on their use of memoization and evaluated on multiple randomly selected inputs from well-known open source input repositories that take at least 30 seconds to run. Table 6.3 also shows the programming effort in number

Program	Description	Total LOC	Source Changes			Profiling Overhead
			# Annot.	Additions/Mods		
				LOC	Desc	
minisat	Boolean Satisfiability Solver	2343	12	36	CC, C, CB	8.4x
ga	Optimization via genetic algorithms	811	8	114	CPP, C, P	8.6x
qmaxsat	Partial MAX Satisfiability solver	1783	12	49	CC, C, AO	10.7x
bobcat	Alpha-beta search based game engine	5255	8	47	CC, C, P	4.9x
ubqp	Binary quadratic program solver	1387	8	81	CC, C, P	15x

Table 6.3: Applications evaluated using WEAKC, total lines of source code, number of annotations, additional changes to source to enable WEAKC parallelization (CC: Copy constructor, C: Comparison operator, CB: Callback, CPP: C++ conversion, P: Progress Measure, AO: Assignment operator) and overhead of profiling over sequential execution.

of WEAKC annotations added and additional changes for implementing standard object oriented abstractions relevant to WEAKC. These changes only introduce sequential code and use no parallel constructs. Safe addition of WEAKC annotations relies on correctly asserting that memoized values within an application can be safely discarded at any time without affecting its correctness, a property that follows from knowledge of application semantics. In all programs, this property was determined after a few hours of studying the baseline sequential implementations.

In addition to WEAKC, two most related non-WEAKC semantic parallelization schemes were evaluated for comparison: Privatization [54, 79] and Complete-Sharing [40, 128, 179]. Both these schemes have different (non-adaptive) synchronization methods as described in Section 5.3, but are based on the same POSIX-based parallel subsystem as WEAKC. Figure 6.9 shows detailed performance results for `minisat` and Figure 6.10 shows all other results. The evaluation was done on a 1.6Ghz Intel Xeon 64-bit dual-socket quad core machine with 8GB RAM running Linux 2.6.24.

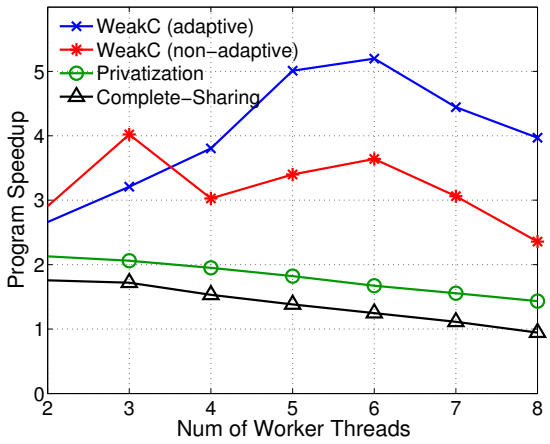
Program	Input Source	# Inps.	Best Speedup/ Quality	Gain over best	
				non-adaptive	non-WEAKC
minisat	Sat-Race	21	5.2x	29.3%	148.0%
ga	HPEAC	22	3.4x	42.6%	86.1%
qmaxsat	Max-SAT	20	2.6x	24.8%	22.8%
bobcat	EndGDB	10	3.2x	14.1%	54.2%
ubqp	ACO-Set	15	11%	1.5%	3.8%

Table 6.4: The input repositories for the evaluated programs, number of inputs, best speedup/quality of result achieved and the performance improvement of adaptive-WEAKC over other best schemes.

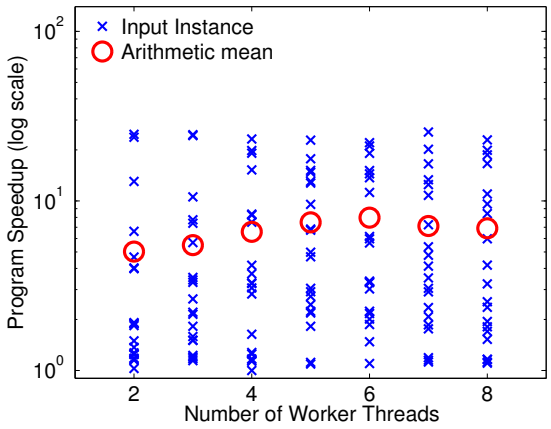
6.2.1 Boolean satisfiability solver: `minisat`

A total of 12 annotations are inserted to annotate (a) `learnts` as a weakly consistent set, (b) `ClauseAllocator` class as an allocator for elements of the weakly consistent set (c) `progressEstimate` member function, which computes an approximate value for search progress using current search depth and number of solved clauses and assigned variables, as a progress measure. Additionally, a copy constructor was added to `Solver` to enable WEAKC runtime to perform initial solver replication; an overloaded comparison operator that uses activity heuristics within the solver to rank weakly consistent set elements, and two callbacks to interface garbage collected elements with the main `Solver` state.

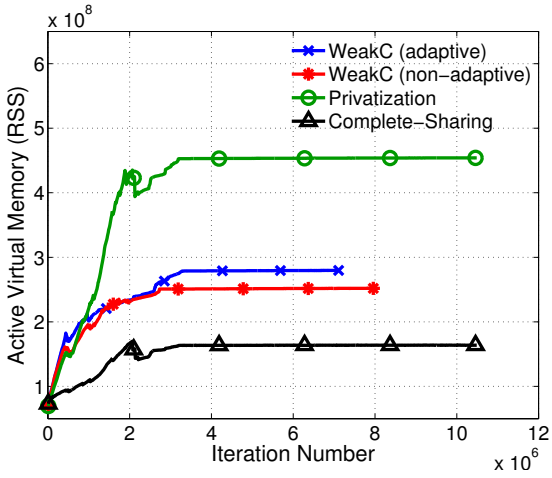
Speedup and Variance. Figure 6.9(a) shows the speedup graph for `minisat`. The adaptive version of WEAKC outperforms the rest by a wide margin. It scales up to six worker threads (a total of twelve POSIX threads, with six additional tuning threads) achieving a geomean program speedup of 5.2x over sequential, after which the speedup decreases mainly due to cache interference between multiple parallel workers (14% increase in L2 data cache miss rate from 6 to 7/8 threads). The non-adaptive WEAKC version achieves better speedup than adaptive WEAKC for up to three worker threads but slows down beyond that point. Both Privatization and Complete-Sharing versions show poor scaling. In



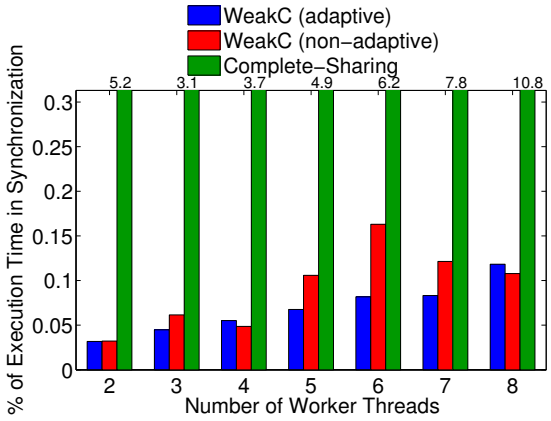
(a) Performance (Geomean Speedup)



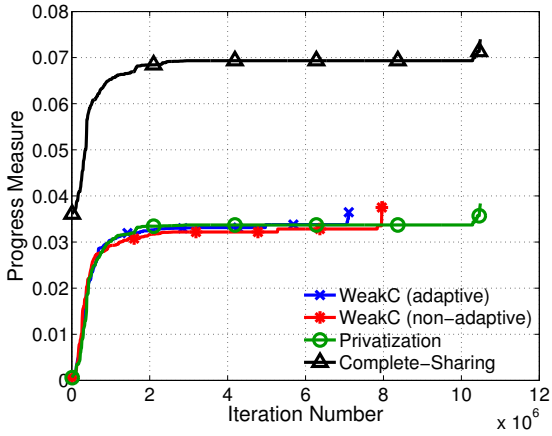
(b) Variations in Speedup (WEAKC-adaptive)



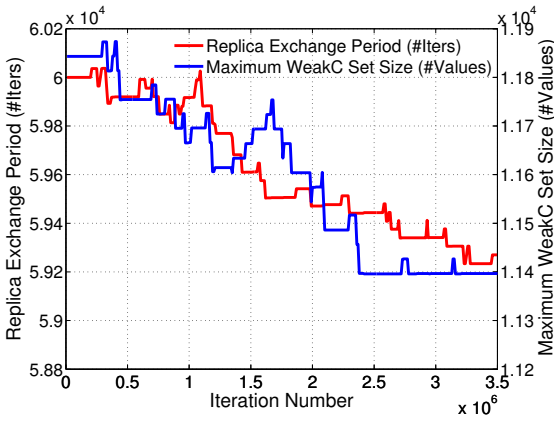
(c) Memory Consumption



(d) Synchronization overhead



(e) Estimate of search progress at runtime



(f) Tuning of WEAKC parameters

Figure 6.9: WEAKC Experimental results for `minisat`

addition to data cache misses, repeated computation of the same learnt clauses across different parallel workers in the former and extremely high synchronization costs in the latter lead to speedup curves with negative slopes. Figure 6.9(b) shows the variations in speedups across different inputs (in log scale) for the adaptive version. The speedups range from a minimum of 1.1x to a maximum of 25x; this high variance demonstrates the sensitivity of SAT execution times to input behavior and consequent usefulness of online adaptation in optimizing parallel configuration at runtime.

Memory Consumption. Figure 6.9(c) shows the active memory consumed during search, sampled and measured using `getrusage()`. Amongst the four schemes, Privatization consumes the maximum amount of memory. Although it starts with a similar memory profile to others as search progresses redundant conflict clause learning by parallel workers increases its memory consumption at a much higher rate than others. The Complete-Sharing scheme consumes the least memory largely due to avoidance of redundant learning. The memory consumption for the WEAKCschemes is in between: the memory consumed by the adaptive version includes that used by tuners threads which, as seen from the graph, has a modest impact on the program's overall memory consumption.

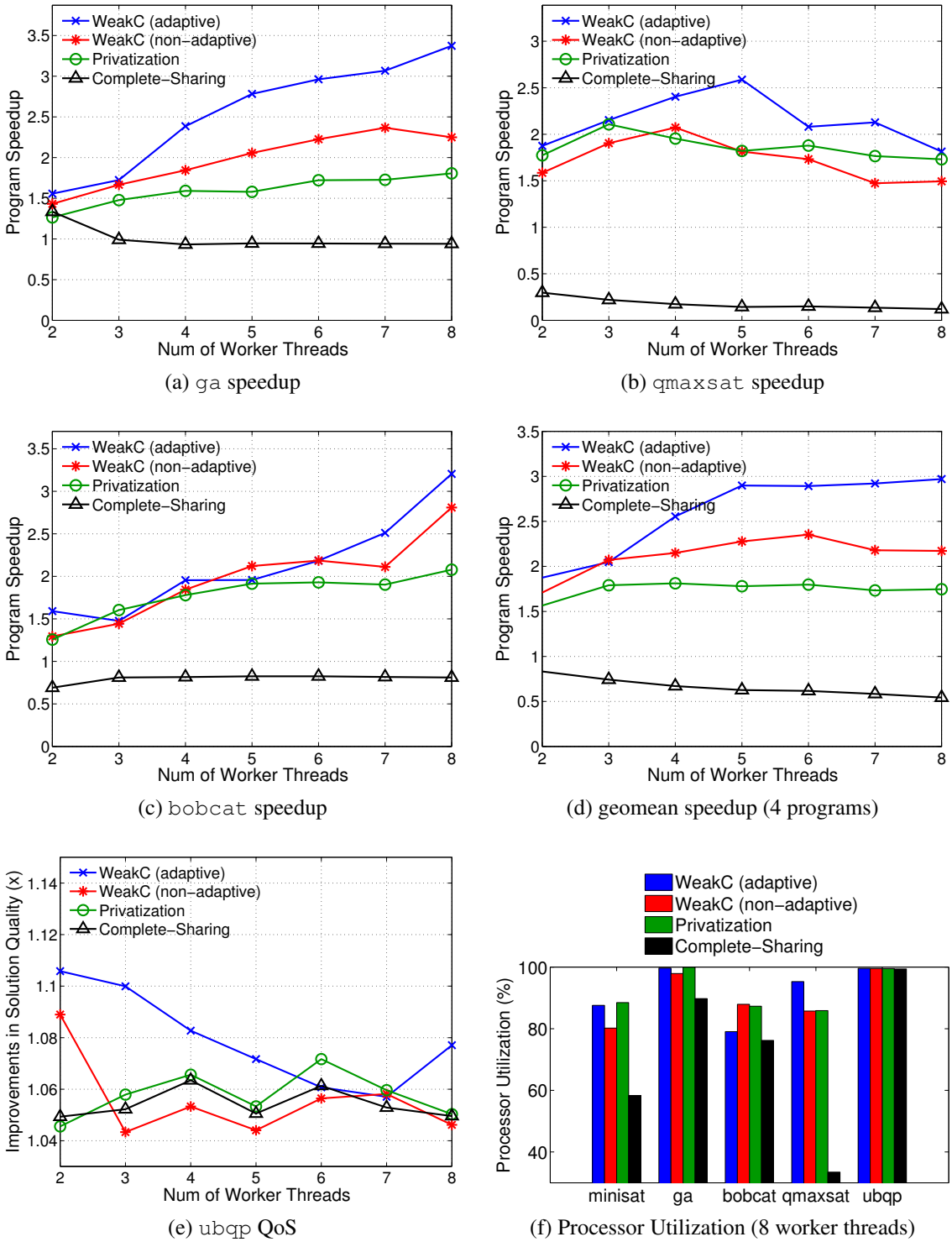
Search Progress and Synchronization. Figure 6.9(d) shows the percent of execution time spent in synchronization. Figure 6.9(e) shows the evolution of progress measured during search. As seen from these graphs, although search progress improves fastest per iteration for Complete-Sharing, the corresponding high synchronization costs result in overall slower convergence. The other schemes make relatively slower progress per iteration than Complete-Sharing. The WEAKC schemes converge the fastest followed by Privatization as seen from the early termination of their progress curves. Given that Complete-Sharing's synchronization costs is an order of magnitude higher than WEAKC, and Privatization has

longer convergence time due to low search space pruning, sparse sharing becomes key to fast convergence times.

Online adaptation. Figure 6.9(f) shows a snapshot of adaptation for two WEAKC parameters at runtime: the replica exchange period and shadow replica set size. The curve for the third parameter, save period, is not shown as it is similar to exchange period offset by a certain factor. Initially, the replica exchange period is high implying a low initial frequency of sharing, but as the search progresses, WEAKC tuning decreases the value of the replica exchange period. The value for shadow replica size is high in the beginning, but with time the number of elements shared decreases. Overall, tuning in `minisat` causes plenty of elements to be shared less frequently during the initial phases of search; as parallel workers start to converge in later phases, tuning causes *fewer* elements to be shared *more* frequently among parallel workers.

6.2.2 Genetic algorithm based Graph Optimization: `ga`

`ga` is a genetic algorithm program originally written in C [93] which we converted to C++. It uses operations like mutation, selection, and crossover to probabilistically create newer generations of candidate solutions, based on fitness scores of individuals in the current generation. Solutions are represented as chromosomes within the program. The search terminates on reaching sufficient fitness for a population. The `ga` program in our evaluation additionally uses the concept of elite chromosomes [140], where a non-deterministic fraction of the fittest individuals within certain thresholds are carried forward across generations. WEAKC annotations were applied to the elite chromosome set. The thresholds on elite chromosomes were automatically enforced by a hard limit defined on the native



(a) `ga` speedup

(b) `qmaxsat` speedup

(c) `bobcat` speedup

(d) geomean speedup (4 programs)

(e) `ubqp` QoS

(f) Processor Utilization (8 worker threads)

Figure 6.10: WEAKC Experimental Results II

chromosome set. The correctness of applying WEAKC annotations follows from the observation that missing a few elite chromosomes can only delay convergence without affecting core algorithm functionality.

On the other hand, sharing the best chromosomes between different parallel workers can potentially speed up convergence. Apart from the annotations for weakly consistent sets, code changes involved conversion of C code into C++. The progress measure returns a scaled fitness value of the best chromosome in the current generation. Figure 6.10(a) shows the speedup graph for `ga`. Adaptive WEAKC scales up to eight worker threads achieving a speedup of 3.4x. Although both non-adaptive WEAKC and Privatization show similar scaling trends, their speedup curves have a much smaller slope than adaptive WEAKC. Finally, Complete-Sharing shows no speedup beyond two threads, with synchronization costs at higher thread counts causing a slight slowdown.

6.2.3 Partial Maximum Satisfiability Solver: `qmaxsat`

`qmaxsat` [124] solves partial MAXSAT, an optimization problem of determining an assignment that *maximizes* the number of satisfied clauses for a given boolean formula. It works by successively invoking a SAT subroutine, each time with a new input instance. Each SAT invocation adds to a growing set of learnt clauses. However, there is a subtle difference in semantics of learnt clauses between `qmaxsat` and `minisat`. In `qmaxsat` input instances change across SAT invocations, so learnt clauses from newer generations have different variables/clauses compared to previous generations. In the context of WEAKC, this means that when parallel workers progress at different speeds, learnt clauses of newer generations belonging to a parallel worker should not be shared and merged with learnt clauses of previous generations belonging to other workers.

The re-entrant and compositional nature of WEAKC guarantees correctness transparently by ensuring that parallel SAT invocations across generations always proceed in lock

step. The potential increase in overheads for thread creation, destruction, solver replication, and re-initialization due to multiple entries into the runtime is avoided by performing parallel independent replication and re-initialization. Figure 6.10(b) shows the speedup graph. The adaptive version peaks at five threads with a geomean speedup of 2.6x. Interestingly, the Privatization scheme performs better than the non-adaptive WEAKC scheme although both start to scale down beyond three threads. The synchronization costs for the Complete-Sharing scheme cause a slowdown at all thread counts. Overall, although `qmaxsat` is reasonably long running, learnt clause sharing and tuning occur within a shorter window for each of the multiple invocations of SAT within the main program, compared to `minisat`. The consequent constrained sharing profile together with the cache sensitive nature of parallel SAT are reflected in the performance.

6.2.4 Alpha-beta search based game engine: `bobcat`

`bobcat` [89] is a chess engine based on alpha-beta search. It uses a hash table called the "transposition table" [187] that memoizes the results of a previously computed sequence of moves to prune the search space of a game tree. Using WEAKC annotations, this table was assigned weakly consistent map semantics. Being a purely auxiliary data structure akin to a cache, partial lookups and weak mutation of the transposition table only cause previously determined positions to be recomputed, without affecting correctness. Apart from adding a copy constructor and a comparison operator that uses the age of a transposition for ranking, the transposition table interface was made generic using C++ templates to expose key and value types. The progress measure employed returns the number of nodes pruned per second weighted by search depth. Figure 6.10(c) shows the speedup graphs. In contrast to `minisat`, the benefits of WEAKC start only after five worker threads, with the adaptive version achieving a best geomean speedup of 3.2x on eight worker threads compared to a best of 2x for Privatization. Compared to the weakly consistent sets in other programs,

the transposition table in `bobcat` is small in size and is accessed with a high frequency. This causes Complete-Sharing to perform poorly, resulting in a 20% slowdown due its associated high synchronization overhead.

6.2.5 Unconstrained Binary Quadratic Program: `ubqp`

`ubqp` [35] solves an unconstrained binary quadratic programming problem using ant colony optimization. The algorithm maintains a population of "ants" that randomly walk the solution space and record history about the fitness of a solution in a "pheromone matrix" structure. Similar to `ga`, elitism within this program holds a collection of fittest solutions within a set, to which we applied WEAKC annotations. Instead of a conventional convergence criterion, the search loop in `ubqp` stops when a given time budget expires. Parallelizing `ubqp` can improve this program not by reducing its execution time, but by improving the quality of the solution obtained within this fixed time frame. Our evaluation measures the improvement in the quality of the final solution (via an application level metric) of parallel execution over sequential when both are run for a fixed time duration. Figure 6.10(e) shows the results. Although for few threads the adaptive version has a better solution quality than the other techniques (11% improvement), the general trend of all the curves is downward. This occurs because although WEAKC enables sharing of fitter solutions among different workers, the pheromone matrix that encodes solutions history is not shared among workers due to high associated communication costs.

6.2.6 Discussion

The application of WEAKC achieves a geomean speedup of 3x on four programs (Figure 6.10) and improves the solution quality for one program by 11%, while the best non-WEAKC scheme (Privatization) obtains a geomean speedup of 1.8x and 7.2% improvement

Program	Scientific Field	Target Function(s)	Exec Time of Target Loops	Total Src LOC	LOC Changed	LOC Change Desc.
WithinHostDynamics	Ecology and Evolutionary Biology	`main`	100%	2415	2	Privatization
packing	Computational Physics	`ForAllNeighbors`	98%	1802	46	Loop Fission
Clusterer	Computational Biology	`Cluster`	99%	18865	-	-
SpectralDrivenCavity	Computational Fluid Dynamics	`main`	66%	840	-	-

Table 6.5: Details of Case Studies of Programs from the Field: the program evaluated, scientific field/domain, target function containing the parallelized loop, execution time of target loops, total source lines of code, lines of code changed and its description

in solution quality, respectively. Regarding programmer effort, an average of 10 WEAKC annotations and 67 sequential lines of code per program are added or modified to implement C++ abstractions related to WEAKC. In spite of creating more threads than the number of available hardware contexts, adaptive WEAKC outperforms other schemes for all evaluated programs. For `ga` and `bobcat`, it attains peak speedup at eight worker threads with sixteen threads in total, outperforming other schemes that create only eight threads in total. For `minisat` and `qmaxsat`, although the peak speedups for adaptive version are obtained at six and five worker threads respectively, their processor utilization is comparable to or better than the other schemes as seen in Figure 6.10(f). This points to a relatively low impact of context switching and synchronization on parallel execution. By contrast, CompleteSharing has much lower processor utilization compared to other schemes, indicating that synchronization takes away a fair share of processor time from useful computation.

Program	Num of COMMSET Annotations	COMMSET attributes	Applicable Parallelizing Transforms	Best Speedup	Best Parallelization Scheme	Num of custom AA rules
WithinHostDynamics	4	I, S, O, C	DOALL, PS-DSWP	6.5x	DOALL + Mutex	9
packing	1	S, C	DOALL	3.1x	DOALL + Mutex	5
Clusterer	1	S, C	DOALL, PS-DSWP	7.4x	DOALL + Spin	3
SpectralDrivenCavity	0	-	DOALL	1.9x	DOALL	0

Table 6.6: Details of Case Study Programs parallelized: the program parallelized, number of lines of commutativity annotations added, the various COMMSET attributes used (PI: Predication at Interface, PC: Predication at Client, C: Commuting Blocks, I: Interface Commutativity, S: Self Commutativity, G: Group Commutativity, O: Optional Commuting Block), applicable parallelizing transforms, the best speedup obtained and the corresponding (best) parallelization scheme, and the number of custom alias disambiguation rules employed

6.3 Case Studies of Programs from the Field

This section describes the evaluation of the IPP techniques developed in this dissertation with some case studies of programs collected during the field study. Tables 6.5 and 6.6 give the details of these programs.

6.3.1 WithinHostDynamics

This program is from the field of Ecology and Evolutionary Biology. As gathered from our field study, the author of this program is interested in studying the within-host competition of parasites and how this impacts epidemiology and control of a disease. This study enables a better understanding of anti-malarial drug resistance, which is a major public health problem that impedes the control of malaria.

The program is written in C++ and performs discrete simulation of within-host dynamics. The simulation is performed within an outer loop that runs for 10000 times, with each

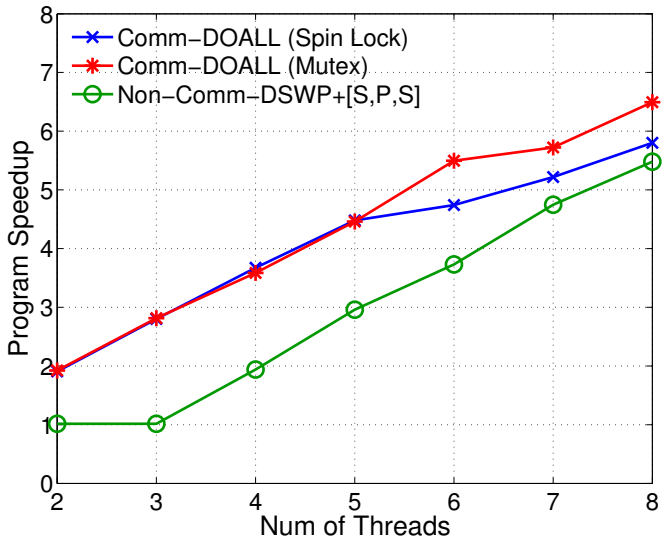


Figure 6.11: Speedup over sequential execution for `WithinHostDynamics`, a program for simulating dynamics of parasitic evolution within a host

iteration that takes about 1 minute to run, resulting in a total of around 6 days for completion. Speeding up this loop can enable the author to gain the ability to create more complex models for spread of malaria that run within the same amount of time.

COMMSET annotations were applied to relax the semantics of the program in two places: (a) To commute a shuffle function transitively called from the main loop with the use of named commuting blocks which are bound to COMMSETs at their call site. This shuffle function internally uses a random number generator (b) To commute console prints and insertion of simulation output into a vector that has set semantics. This commutativity specification was made by use of a atomic self-commuting code block that encapsulated both the functions. With these annotations, a DOALL parallelization was enabled that achieved a peak speedup of 6.5x on eight threads. This compares favorably to a PS-DSWP style parallelization that gave a speedup of 5.4x without any use of annotations. The speedup graphs are shown in Figure 6.11.

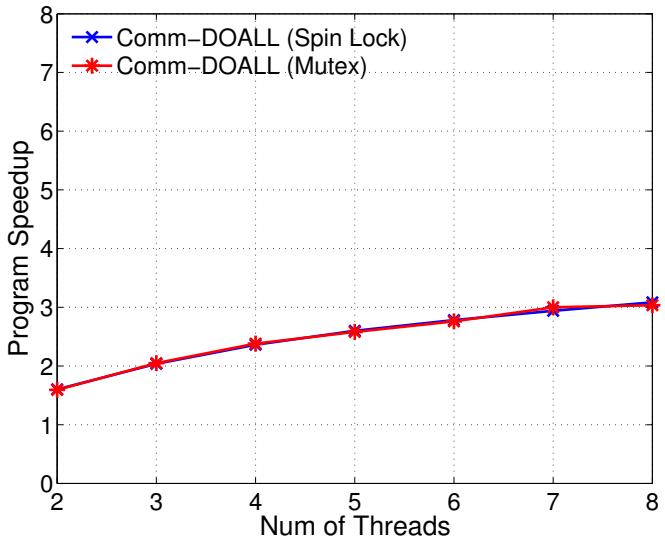


Figure 6.12: Speedup over sequential execution for `Packing`

6.3.2 Packing

This sequential program is from the field of Computational Physics. The author of this program is interested in studying the equilibrium and dynamical properties of a variety of materials which have important applications in nano-engineering. To this end, this program, written in C++, implements a collision-driven packing generation algorithm for studying hard-packings in higher dimensional spaces.

For large inputs, the original sequential code in `packing` takes weeks to produce interesting packings. The author estimates that with accelerated computation, artificial effects due to small system sizes can be removed and additional numerical validation for the author's new theory on particle packings would be produced. Additionally, it would enable research into improving the quality of final jammed packings.

Most of the execution time in the original program is spent in an infinite loop that does iterative refinement of sphere packing. Within this infinite loop, there are three inner loops that are invoked. The first inner loop computes the nearest neighbors for each cell in the sphere and the second inner loop computes offsets from nearest neighbors. The second

inner loop also tests an exit condition for the outer loop. Finally, a third inner loop predicts a collision operation based on the offsets computed from each cell.

The inner loops, as written in the original sequential code, get invoked a large number of times from within the outer loop, but each invocation only executes for a small number of times. The third loop is parallelizable with COMMSET and DOALL parallelization but does not give any speedup due to very low time spent per-invocation. However, it is possible to aggregate different invocations of the first two inner loops across iterations of the outer loop into a larger inner loop followed by an aggregated loop that does collision prediction. With this manual transformation, COMMSET enabled DOALL parallelization gives a speedup 3.1x with `packing`. Specifically, the COMMSET primitive is applied as follows: The third inner loop internally calls a function called `PredictCollision` which has aggregate MAX reduction operations involving floating point operations. These operations are wrapped into an atomic code block and added to a SELF commutative set.

6.3.3 Clusterer

This program is part of the sleipnir [102] tool chain, a popular tool for analyzing genomics data within the field of computational biology. `Clusterer` implements various kinds of hierarchical clustering algorithms using one of sleipnir's different similarity measures.

The target loop for parallelization within `Clusterer` is an inner loop that performs k-means clustering. This loop has irregular floating point MAX reduction operations which are marked are semantically commutative by enclosing them within an atomic commuting block that is added to a SELF COMMSET. With this modification, this loop is parallelizable with DOALL. Synchronization is automatically inserted by the compiler using `pthreads` locks. With eight threads, a speedup of 7.4x is obtained with DOALL and COMMSET. An alternate parallelization that does not require semantic changes is scheduling the MAX

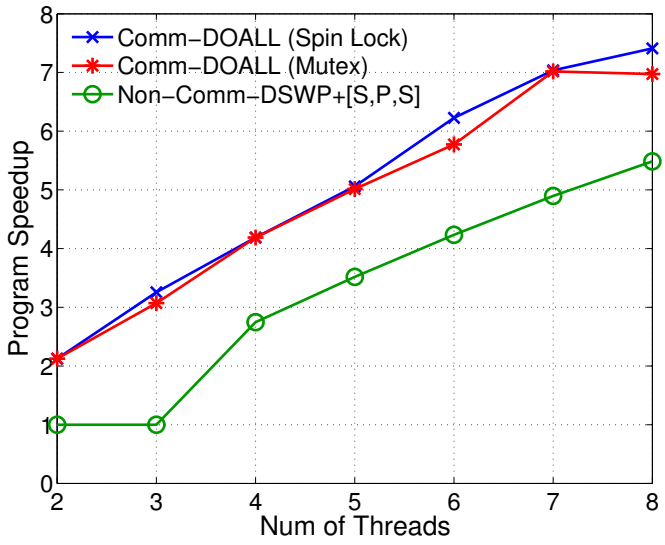


Figure 6.13: Speedup over sequential execution for `Clusterer`, a computational biology program for performing non-hierarchical clustering of gene sequences

operations onto the last sequential stage of a PS-DSWP pipeline. This parallelization results in a peak speedup of 5.5x. Figure 6.13 shows the speedup graphs.

It is interesting to note that the loop structure of the parallelized loop in `Clusterer` is very similar to that of `kmeans` from the STAMP benchmark suite (see Section 6.1.3). However, in `kmeans`, the parallelized loop is tight where the relative execution time of non-commutative code sections reduces drastically with increasing thread counts. This makes the synchronization overhead due to lock contention on commutative blocks more pronounced even for lower thread counts. In contrast, within `Clusterer`, the distribution of execution times is skewed in favor of non-commutative sections that can run in parallel without any synchronization and hence DOALL with COMMSET scales well and performs much better than a PS-DSWP schedule.

This comparison illustrates one of the main benefits of implicit parallel programming via semantic annotations: Even though the loop structures are very similar for the two programs, the optimal parallelization scheme is very different and depends a wide range of machine and input parameters. Relying on the programmer to express relaxed semantics

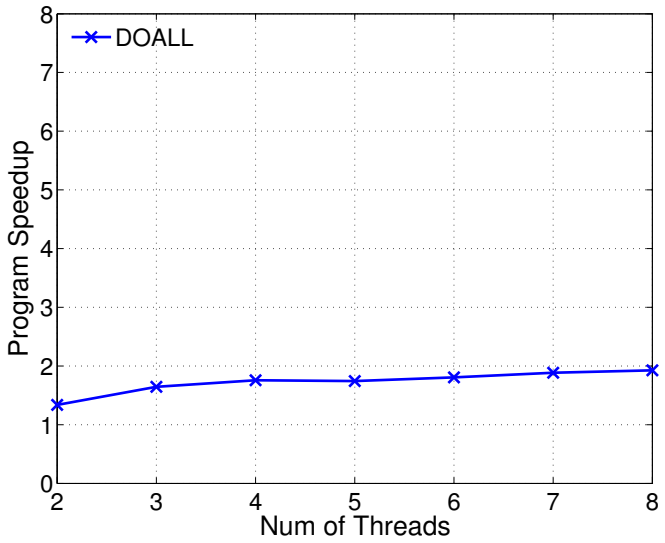


Figure 6.14: Speedup over sequential execution for `SpectralDrivenCavity`

and leaving the "how" of parallelization to rest of the tool chain presents a way to optimally extract thread level parallelism while still preserving performance portability and abstraction boundaries between software and hardware.

The above program was also run two machines that are normally utilized by a scientist-user of this program. The first is a 2GHz Intel Xeon 64-bit eight-core machine with 4GB RAM running Linux 2.6.18. The second is a 2.53GHz Intel Xeon 64-bit sixteen-core machine with 8GB RAM. On the first machine, a speedup of 5.7x was recorded while on the second machine a speedup of 7.38x was recorded. Both these speedups were produced by DOALL parallelization after COMMSET annotations were applied.

6.3.4 SpectralDrivenCavity

This program implements the spectral method for numerically solving Navier-Stokes equations. It is used within the field of Computational Fluid Dynamics. The program in written in C and the main outer loop is a time-step based loop that performs within its various inner loops a $X_{i+1} = f(X_i)$ operation for various instances of X, where each X is a matrix.

Three of these inner loops were automatically parallelizable without any semantic change with the DOALL transformation. The loops together constitute around 66% of execution time. A peak speedup of 1.9x is obtained on 8 threads. Figure 6.14 shows the speedup graphs. The main reason for the limited speedup is that each of the inner loops get invoked with a high frequency within the outer time step loop while each inner loop execution by itself relatively runs for a fairly few number of iterations. SpectralCavity was also run on a machine that is normally used by a scientist-user of this program. The machine is a 3.4GHz Intel Core i7 eight-core machine with 8GB RAM running Linux 2.6.32. A speedup of 1.6x was recorded on this machine without any additional changes on this machine.

Chapter 7

Related Work

This chapter presents a summary of prior work related to the field study, and parallelization systems that leverage techniques that are related to semantic commutativity and weakly consistent data structures and provides an in-depth comparison with the techniques presented in this dissertation.

7.1 Prior studies of the practice of computational science

Hannay et al. [94, 145] conducted an online survey of two-thousand scientists with a goal of studying the software engineering aspects of scientific computation, from a correctness and program development perspective. The survey was carried out anonymously and conclusions drawn without knowledge of the subject's computing environment. Cook et al. [56, 159] present a survey to exclusively understand the programming practices and concerns of potential parallel programmers. The subjects involved in this survey were attendees of a supercomputing conference (SC 1994), and the majority of subjects were computer scientists.

In contrast, the field study presented in this dissertation involved scientists working in diverse fields, and was conducted within the framework of a known (university) scientific computing ecosystem. This field study was carried out in the form of a detailed interview with the subjects. While also covering the correctness and software engineering aspects of computational science, the field study presented in this chapter primarily addressed the need for computational performance, and the practices followed by researchers in enhancing performance in their research.

7.2 Research related to Commutative Set

Semantic Commutativity based Parallelizing Systems. Jade [179] supports object-level commuting assertions to specify commutativity between every pair of operations on an object. Additionally, Jade exploits programmer written read/write specifications for exploiting task and pipeline parallelism. The COMMSET solution relies on static analysis to avoid read/write specifications and runtime profiles to select loops for parallelization. Galois [128], a runtime system for optimistic parallelism, leverages commutativity assertions on method interfaces. It requires programmers to use special set abstractions with non-standard semantics to enable data parallelism. The COMMSET compiler currently does not implement runtime checking of COMMSETPREDICATEs required for optimistic parallelism. However, the COMMSET model is able to extract both data and pipelined parallelism without requiring any additional programming extensions.

DPJ [38], an explicitly parallel extension of Java uses commutativity annotations at function interfaces to override restrictions placed by the type and effect system of Java. Several researchers have also applied commutativity properties for semantic concurrency control in explicitly parallel settings [46, 125]. Paralax [205] and VELOCITY [40, 41]

exploit self-commutativity at the interface level to enable pipelined parallelization. VELOCITY also provides special semantics for commutativity between pairs of memory allocation routines for use in speculative parallelization. Compared to these approaches, the COMMSET language extension provides richer commutativity expressions. Extending the compiler with a speculative system to support all COMMSET features at runtime is part of future work.

Tables 4.1 and 4.2 summarize the relationship between COMMSET and the above programming models.

Commutativity and Concurrency Control. Weihl [211] proposed the use of commutativity conditions for concurrency control of abstract data types. Koskinen et al. [125] use commutativity properties for conflict detection in transactional memory systems. Carlstrom et al. [46] build transactional collection classes with support for semantic concurrency control. Herlihy et al. [98] leverage commutativity to transform linearizable objects into high performing transactional objects. Kulkarni et al. [127] propose the idea of a commutativity lattice to enable the automatic synthesis of different synchronization schemes with varying degrees of granularity. In our work, we developed abstractions for succinctly specifying semantic commutativity properties for enabling compiler driven parallelization and concurrency control is one part of the parallelization scheme.

Checking Semantic Commutativity. Various proposals have been made for checking semantic commutativity conditions. Kim et al. [120] present a framework for logical specification of various kinds of commutativity conditions on well known data structure operations and provide techniques to verify these conditions using a program verification system. Burnim et al. [42] propose semantic atomicity, a property similar to semantic commutativity but applied in the context of multithreaded programs, and present a framework based

on bridge predicates to specify and verify this property. Elmas et al. [74] present methods for verification of shared memory multithreaded programs using the concepts of abstraction and reduction. Based on this, they propose an annotation assistant [75] for interactive debugging of synchronization idioms. The COMMSET system can be augmented by the above techniques to formally prove the correctness of the IPP parallelizations presented in this dissertation.

Implicit Parallel Programming. OpenMP [24] extensions require programmers to explicitly specify parallelization strategy and concurrency control using additional primitives ("`#pragma omp for`", "`#pragma omp task`", "`critical`", "`barrier`", etc.). In the COMMSET model, the choice of parallelization strategy and concurrency control is left to the compiler. This not only frees the programmer from having to worry about low-level parallelization details, but also promotes performance portability. Implicit parallelism in functional languages have been studied recently [95]. IPOT [208] exploits semantic annotations on data to enable parallelization. Hwu et al. [104] also propose annotations on data to enable implicit parallelism. COMMSET extensions are applied to code rather than data, and some annotations like `reduction` proposed in IPOT can be easily integrated with COMMSET.

OOOJava [106] presents a system which allows a programmer to reorder code blocks using annotations. These annotations are analyzed and exploited to produce safe, deterministic parallelism. Compared to COMMSET, the annotation language is not as expressive and does not exploit relaxable semantics that is semantically deterministic.

The PetaBricks [19] language implementation has an autotuning framework that exploits declaratively specified algorithmic choices to yield scalable performance. Parcae [171] is a system for platform-wide dynamic tuning that includes a parallelizing compiler and an autotuning runtime system. The COMMSET system presented in this dissertation has been

integrated with Parcae, and many of the COMMSET programs evaluated in Section 6.1 have been autotuned.

Compiler Parallelization. Research on parallelizing FORTRAN loops with regular memory accesses [73, 118] in the past has been complemented by more recent work on irregular programs. The container-aware [216] compiler transformations parallelize loops with repeated patterns of collections usage. Existing versions of these transforms preserve sequential semantics. The COMMSET compiler can be extended to support these and other parallelizing transforms without changes to the language extension.

Commutativity Analysis. Rinard et al. [180] proposed a static analysis that determines if commuting two method calls preserves concrete memory state. Aleen et al. [16] apply random interpretation to probabilistically determine function calls that commute subject to the preservation of sequential I/O semantics. Programmer written commutativity assertions are more general since they allow multiple legal outcomes and give a programmer more flexibility to express intended semantics within a sequential setting.

7.3 Research related to WEAKC

Explicit parallelization of search and optimization. Existing parallelizations of search and optimization algorithms are pre-dominantly based on explicit parallelism. For instance, SAT solvers have been parallelized for shared memory [91] and clusters [84] using threading and message passing libraries. WEAKC, in contrast, presents semantic language extensions to the sequential programming model promoting easy targeting to multiple parallel substrates without additional programming effort. Additionally, WEAKC performs online adaptation of parallelized search programs.

Smart, distributed, and concurrent data structures. Smart data structures [72] employ online machine learning to optimize throughput of their concurrent data structure operations. STAPL [176] is a parallel version of STL that uses an adaptive runtime for performance portability. Unlike WEAKC, both these libraries preserve semantics of corresponding sequential data structures. Another difference is that online adaptation in WEAKC is done in conjunction with application level performance metrics (search progress) to optimize overall performance and not only to improve data structure throughput. Chakrabarti et al. [48] present distributed data structures with weak semantics within the context of parallelizing a symbolic algebra application. Compared to WEAKC, these data structures require programmers to explicitly coordinate local and remote data transfers, and its semantics enforces eventual reconciliation of mutated data across all parallel workers. WeakHashMap [70] has semantics similar to WEAKC's data structures, but unlike WEAKC it only supports strong deletion and requires programmers to explicitly insert synchronization and coordinate parallel data access. Kirsch et al. [161] propose k-FIFO queue, a concurrent queue that allows semantic deviations from sequential order for up to k elements.

Asynchronous Iterative Algorithms (AIA). Similar to ALTER discussed in Table 1, asynchronous iterative algorithms [138, 146, 121] are certain types of stencil computations that are immune to bounded levels of stale reads of scalar values in a parallel setting. However, they still need to enforce periodic or eventual reconciliation of stale values for correctness. Compared to AIA, WEAKC's consistency property applies to compound data types and is much weaker: the throwaway, temporally-agnostic and reusable nature of memoized data in WEAKC ensuring correct application behavior without needing reconciliation. The nature of algorithms targeted by our WEAKC implementation, viz. combinatorial search and optimization utilizing memoization is different from AIA.

Adaptive multicore caching. Hardware caches and weakly consistent data structures are both performance critical structures having data that can be discarded at any time without affecting correctness. Recent work has looked at adapting shared multicore caching policies to varying workloads. Beckmann et al. [30] present an adaptive data replication scheme within a private-shared multicore caching system for exploiting low latency of private caches without incurring high capacity misses. Bitirgen et al. [33] apply online machine learning to adaptively regulate shared cache partitioning within a wider hardware resource allocation framework. Qureshi et al. [168] use dynamic cache miss rates induced by each application to adaptively partition a shared cache between multiple concurrently running applications. The sharing and adaptation in WEAKC in many ways resemble those in multicore caches, although at a much coarser-grain, semantic application level.

Memory consistency models. Various notions of weak consistency have been studied in the context of multiprocessor memory models [13]. A memory model depicts the order in which load/store operations to memory locations appear to execute, and addresses the question: "What value can a load of a *memory location* return?". By contrast, WEAKC is concerned with the order and semantics of high-level data structure operations, and addresses "What values can a *data structure query* return?". A concurrent implementation of weakly consistent data structure (as realized in WEAKC) can be achieved on processors with either a sequentially or weakly consistent memory model, with conformance to data structure semantics ensured by appropriate use of low level atomics.

Compiler and runtime support for application adaptation. Adve et al. [14] propose compiler and runtime support for adaptation of distributed applications. AARTS [199] is a lightweight auto-tuning framework for optimizing nested loop parallelism. Both these systems rely on programmers to explicitly parallelize programs, select tuning parameters,

and insert runtime calls for tuning at profitable points in application code. Active Harmony [197] provides a runtime API for programmers to expose tunable parameters and specify optimization metrics. The runtime automatically monitors the tunable parameters and applies a variety of online optimization algorithms for tuning. Rinard et al. [181] present a parallelizing compiler that performs adaptive runtime replication of data objects to minimize synchronization overhead. Parcae [171] presents an automatic system for platform-wide dynamic tuning. Compared to these systems, WEAKC exploits the semantics of weakened consistency data structures to optimize parallel configuration at runtime by automatically selecting and tuning parameters implicitly exposed by this weakening.

Chapter 8

Future Directions and Conclusions

As the field study in this dissertation shows, we are in an era where the twin trends of data-driven scientific research and ubiquitous parallel computing are converging at a rapid pace. In this situation, it is vitally important to synergistically exploit this convergence to accelerate the pace at which scientific research is conducted. For this to happen, programming techniques and supporting tools should require minimal effort from a scientist-programmer while at the same time delivering on scalable performance on a wide variety of platforms. By preserving the ease of sequential programming and providing natural extensions that require only specific domain inputs from a scientist-programmer, implicit parallel programming provides a promising approach inclined in this direction. This dissertation proposed and evaluated two new semantic language extensions and associated compiler and runtime technology within this area, taking a few more steps toward realizing the full potential of implicit parallel programming for accelerating scientific research.

8.1 Future Directions

This section discusses some avenues for future research related to the implicit parallel programming approach based on the two techniques proposed in this dissertation.

Approximate Parallel Programming. Similar to search and optimization loops that motivated the design and implementation of weakly consistent sequential data structures in WEAKC, several real world scientific applications have hot program loops that iterate to convergence based on pre-determined statistical error bounds. Programmers often make a static fixed choice of these error bounds, which remain unchanged even when running the program on multiple platforms. Static choices for error bounds often result performance unportable programs.

Implicit in this static choice of error-bounds is the trade-off between accuracy of the computation and performance of the corresponding program. One important class of applications where this tradeoff is evident is when performing improving computations, i.e., computations that result in monotonically increasing values, with intermediate results being more approximate than the final result. A classic example occurs in web search [25] and in "big data" [64] computations.

Similar in spirit to WEAKC generalized IPP extensions that permit a programmer to not only declaratively specify a relaxable policy involving the tolerable error thresholds in computation but also a desired performance given a time budget can achieve performance portability by enabling automatic exploitation of the accuracy-performance tradeoff. As in WEAKC, the declarative specification of parameters can enable tools to delay the selection of an optimally performing parallelization or synchronization scheme late in a program's life cycle, typically until complete details of a target parallel architecture and the environment (e.g, input data) are available (either at compile time or runtime).

Implicit Parallel Programming and Heterogeneous Parallelism. Increasingly, heterogeneity is becoming a norm in the world of computing, with the ubiquity of computing substrates designed with completely different optimization criteria interacting in disparate and interesting ways. The emergence of GPUs and multicore onto mainstream computing on both desktops and mobile smartphones and large scale distributed systems that constitute "the cloud" and their interactions presents interesting use-cases for systems researchers. For instance, a system where a mobile CPU interacts with a GPU on the same device and a cloud based backend at a remote location presents endless possibilities for solving computationally hard problems in a parallel and a distributed way. Doing so in a manner that provides a seamless end-user experience requires design of abstractions that are simple and at the same time architecture agnostic at the language level, and at the backend – careful allocation and management of the multiple levels of parallelism exposed by heterogeneous components and optimally balancing tradeoffs between locality, power, and performance. The design choices made in carefully separating the concerns between programmer, compiler, and runtime in parallelization systems presented in this dissertation can serve as an interesting starting point to explore the space of heterogeneous parallel systems in interesting ways.

Synergistic combination with speculative parallelism. Bridges et al [41] demonstrate the utility of combining different forms of speculation, viz. memory, control, alias speculation with commutativity. Extending the different features of COMMSET to work seamlessly with speculative parallelism introduces new opportunities and challenges. In particular, speculatively executing members of a COMMSET while preserving intra-COMMSET atomicity and avoiding unnecessary bookkeeping overheads due to inter-COMMSET synchronization, value speculating the results of COMMSETPREDICATEs when they cannot be resolved statically, deciding whether it is better to execute optionally commuting blocks

sequentially when the speculative validation and atomicity overheads are high are some of the aspects worth investigating. Similarly, elements of speculation could potentially enhance the WEAKC solution. In particular, speculative synchronization [141] can potentially lower the synchronization costs in a way that permits more frequent exchange of weakly consistent data between parallel workers.

Formalizing a general parallelization framework for relaxed programs. In this dissertation, two programming extensions were proposed, one of which was code-centric (COMMSET) and another data-centric (WEAKC). Each had its own associated parallelization solution that was successful in parallelizing a distinct set of applications. It would be interesting to investigate the feasibility of a general formal framework that is based on a set of minimal semantic properties which are composable in different ways leading to a continuum of relaxations of sequential programs along with their associated benefits for parallelization. When combined with existing parallelization strategies and synchronization mechanisms, proving whether specific implementations (e.g, optimistic vs pessimistic synchronization) would always perform better than other implementations can have many benefits, including giving useful deployment advice to a systems practitioner and also enabling certain optimizations to be done statically while deferring the others to until execution. Rinard et al. [44] have done some initial theoretical work along this direction, though it would be interesting to see how the practical solutions proposed and demonstrated in this dissertation on important class of scientific applications relate to such frameworks and exploit the relation therein.

An interesting idea is identifying and generalizing algorithmic patterns currently specific to a particular domain to guide design of general parallelization solutions that across

multiple application areas. For instance, the field study found that interactive data languages are popular for data visualization in astrophysics, primitives for describing graphical models are often employed in political science, and custom solutions exist for audio and video processing in the field of music. Exploiting such patterns for parallelism on multicore has a potential for huge impact but introduces new challenges: the identification of algorithmic idioms within each domain that are amenable to parallelization, and design of systems that allow easy expression of these idioms while optimally exploiting their semantics for delivering on performance.

8.2 Summary and Conclusions

This dissertation presented a field study of the practice of computational science. Overall, this study reveals that current programming systems and tools do not meet the needs of computational scientists. Most tools assume the programmer will invest time and energy to master a particular system. By contrast, scientists tend to want results immediately. Nevertheless, the study discovered that virtually all scientists understand the importance of scientific computing, and many spend enormous time and effort programming. Despite this effort, most scientists are unsatisfied with the speed of their programs and believe that performance improvements will significantly improve their research. In many cases, scientists said that increased performance would not just improve accuracy and allow for larger experiments, but would enable fundamentally new research avenues.

Based on the results of this field study, this dissertation proposed two new implicit parallel programming models called CommSet and WeakC, described their design and implementation within a parallelizing compiler, and evaluated both on real hardware and on several real-world applications. Both these solutions preserve the ease of sequential

programming and require modest effort from a programmer for creating well-performing parallelizations.

COMMSET is an implicit parallel programming model that has a unified, syntactically succinct and generalized semantic commutativity construct. The model provides programmers the flexibility to specify commutativity relations between arbitrary structured blocks of code and does not require the use of any additional parallel constructs. Parallelism exposed implicitly using COMMSET is independent of any particular parallelization strategy or concurrency control mechanism.

The evaluation of the end-to-end COMMSET parallelization system on a set of twenty programs shows that it achieves scalable performance with modest programming effort, and it increases the applicability of existing automatic parallelization techniques. With a total of 38 lines of non-semantic code changes and 78 commutativity annotations over 102,232 lines of source code, an overall geomean speedup of 4x is obtained.

Within the set of twenty programs, COMMSET extensions and associated compiler technology enable scalable parallelization of eight programs that do not scale with existing automatic parallelization techniques, obtaining a geomean speedup of 5.1x. In addition, COMMSET has also been applied to parallelize four programs used by scientists in their day-to-day research that were identified during the field study.

WEAKC is a framework for parallelizing search loops with auxiliary data structures that have weak consistency semantics. WEAKC provides language extensions for expressing weak semantics, and a compiler-runtime system that leverages this semantics for parallelization and adaptive runtime optimization. Evaluation on eight cores shows that WEAKC obtains a geomean speedup of 3x on four programs over sequential execution, and an 11% improvement in solution quality of a fifth program having fixed execution time, compared to 1.8x and 7.2% respectively for the best non-WEAKC parallelization.

In conclusion, this thesis demonstrates that a semantic approach to parallelization has the twin advantages of preserving the ease of sequential programming while overcoming the artificial impediments faced by automatic parallelization in obtaining scalable performance. Given that the importance of leveraging parallelization in scientific research is only set to increase in the era of exponential data growth and increasingly heterogeneous architectures, the approaches described in this thesis in preserving the abstraction boundaries between software and hardware while still resulting in performance portable programs will only become more relevant.

Bibliography

[1] Gordon Bell Prize for Peak Performance 2003. http://www.sc-conference.org/sc2003/tech_awards.html.

[2] Interactive Data Language Online Help. http://idlastro.gsfc.nasa.gov/idl_html_help/home.html.

[3] Max/MSP: An interactive graphical dataflow programming environment for audio, video and graphical processing. http://www.cycling74.com.

[4] Princeton Plasma Physics Laboratory. http://www.pppl.gov/.

[5] School of Engineering and Applied Science (SEAS). http://www.princeton.edu/engineering/.

[6] Terascale Infrastructure for Groundbreaking Research in Engineering and Science. http://tigress.princeton.edu/.

[7] The Carnegie Classification of Institutions of Higher Education. http://classifications.carnegiefoundation.org/.

[8] The Lewis-Sigler Institute for Integrative Genomics. http://www.princeton.edu/genomics/.

[9] The Princeton Institute for Computational Science and Engineering. http://www.picscie.princeton.edu/.

[10] The Princeton Institute for Science and Technology of Materials (PRISM). http://www.princeton.edu/prism/.

[11] Genetic Sequence Data Bank NCBI-GenBank Flat File Release 193, 2012. ftp://ftp.ncbi.nih.gov/genbank/gbrel.txt.

[12] Intel Advisor XE, 2013. http://software.intel.com/en-us/intel-advisor-xe.

[13] S. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *Computer*, 29(12):66–76, 1996.

[14] V. Adve, V. V. Lam, and B. Ensink. Language and Compiler Support for Adaptive Distributed Applications. In *Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, LCTES '01, pages 238–246, New York, NY, USA, 2001. ACM.

[15] S. Agostinelli, J. Allison, K. Amako, J. Apostolakis, H. Araujo, P. Arce, M. Asai, D. Axen, S. Banerjee, G. Barrand, et al. Geant4-A simulation toolkit. *Nuclear Instruments and Methods in Physics Research-Section A Only*, 506(3):250–303, 2003. http://www.geant4.org/.

[16] F. Aleen and N. Clark. Commutativity Analysis for Software Parallelization: Letting Program Transformations See the Big Picture. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[17] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[18] J. Anderson, A. Keys, C. Phillips, T. Dac Nguyen, and S. Glotzer. HOOMD-blue, general-purpose many-body dynamics on the GPU. *Bulletin of the American Physical Society*, 55, 2010.

[19] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: a language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM.

[20] Apple Open Source. md5sum: Message Digest 5 computation. `http://www.opensource.apple.com/darwinsource/`.

[21] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent programming in ERLANG*, volume 2. Prentice Hall, 1996.

[22] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, et al. The Landscape of Parallel Computing Research: A view from Berkeley. Technical report, UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[23] J. S. Auerbach, D. F. Bacon, P. Cheng, and R. M. Rabbah. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. In *OOPSLA*, pages 89–108, 2010.

[24] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems*, 2009.

[25] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 198–209, New York, NY, USA, 2010. ACM.

[26] W. Baek, C. Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun. The OpenTM transactional application programming interface. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, pages 376–387. IEEE, 2007.

[27] V. Balasundaram et al. *Interactive parallelization of numerical scientific programs*. PhD thesis, Rice University, 1989.

[28] T. Ball and J. R. Larus. Optimally Profiling and Tracing Programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[29] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 509–520, New York, NY, USA, 2012. ACM.

[30] B. M. Beckmann, M. R. Marty, and D. A. Wood. ASR: Adaptive Selective Replication for CMP Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 443–454, Washington, DC, USA, 2006. IEEE Computer Society.

[31] D. Benson, M. Boguski, D. Lipman, and J. Ostell. GenBank. *Nucleic acids research*, 25(1):1–6, 1997.

[32] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *Proceedings of the 24th ACM SIGPLAN conference on Object*

oriented programming systems languages and applications, OOPSLA '09, pages 81–96, New York, NY, USA, 2009. ACM.

[33] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 318–329, Washington, DC, USA, 2008. IEEE Computer Society.

[34] G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of NESL. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 1996.

[35] C. Blum and M. Dorigo. The Hyper-Cube Framework for Ant Colony Optimization. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 34(2):1161–1172, 2004.

[36] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The Next Generation in Parallelizing Compilers. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, pages 10–1. Springer-Verlag, Berlin/Heidelberg, August 1994.

[37] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 97–116, New York, NY, USA, 2009. ACM.

[38] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System

for Deterministic Parallel Java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2009.

[39] D. Bolme, M. Strout, and J. Beveridge. Faceperf: Benchmarks for face recognition algorithms. In *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, pages 114–119. IEEE, 2007.

[40] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the Sequential Programming Model for Multi-Core. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 69–84, Washington, DC, USA, 2007. IEEE Computer Society.

[41] M. J. Bridges. *The VELOCITY Compiler: Extracting Efficient Multicore Execution from Legacy Sequential Codes*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, November 2008.

[42] J. Burnim, G. Necula, and K. Sen. Specifying and checking semantic atomicity for multithreaded programs. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XVI, pages 79–90, New York, NY, USA, 2011. ACM.

[43] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[44] M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Reasoning about relaxed programs. 2011.

[45] M. C. Carlisle. *Olden: Parallelizing programs with dynamic data structures on distributed-memory machines*. PhD thesis, 1996.

[46] B. D. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun. Transactional Collection Classes. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2007.

[47] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–13, New York, NY, USA, 2006. ACM Press.

[48] S. Chakrabarti and K. A. Yelick. Distributed Data Structures and Algorithms for Gröbner Basis Computation. *Lisp and Symbolic Computation*, 7(2-3):147–172, 1994.

[49] M. M. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18. ACM, 2007.

[50] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 363–375, New York, NY, USA, 2010. ACM.

[51] R. Chandra. *Parallel programming in OpenMP*. Morgan Kaufmann, 2001.

[52] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[53] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. 2009.

[54] R. Cledat, T. Kumar, and S. Pande. Efficiently Speeding up Sequential Computation through the N-way Programming Model. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, New York, NY, USA, 2011. ACM.

[55] E. M. B. Consortium et al. EEMBC benchmark suite, 2009.

[56] C. R. Cook, C. M. Pancake, and R. A. Walpole. Are expectations for parallelism too high? A survey of potential parallel users. In *Proc. of SC*, pages 126–133, 1994.

[57] D. Coppersmith. The Data Encryption Standard (DES) and its strength against attacks. *IBM Journal of Research and Development*, 38(3):243–250, 1994.

[58] R. W. Cox. AFNI: Software for analysis and visualization of functional MRI. *Computers and Biomedical Research*, 29:162–173, 1996. `http://afni.nimh.nih.gov/afni`.

[59] R. Cytron. DOACROSS: Beyond vectorization for multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 836–884, August 1986.

[60] A. Daniel. CLOC: Count lines of code, 2006. `http://cloc.sourceforge.net/`.

[61] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel Programming Using Skeleton Functions. In *Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*, PARLE '93, pages 146–160, London, UK, UK, 1993. Springer-Verlag.

[62] J. R. B. Davies. Parallel loop constructs for multiprocessors. Master's thesis, Department of Computer Science, University of Illinois, Urbana, IL, May 1981.

[63] M. de Hoon, B. Chapman, and I. Friedberg. Bioinformatics and computational biology with Biopython. *Genome Informatics Series*, pages 298–299, 2003.

[64] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[65] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[66] P. J. Denning. Computer science. In *Encyclopedia of Computer Science*, pages 405–419. John Wiley and Sons Ltd., Chichester, UK.

[67] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: a domain specific language for building portable mesh-based PDE solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[68] D. Dig. A refactoring approach to parallelism. *Software, IEEE*, 28(1):17–22, 2011.

[69] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 223–234, New York, NY, USA, 2007. ACM.

[70] K. Donnelly, J. J. Hallett, and A. Kfoury. Formal semantics of weak references. In *Proceedings of the 5th international symposium on Memory management*, ISMM '06, pages 126–137, New York, NY, USA, 2006. ACM.

[71] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, July 1999. `http://hmmer.janelia.org/`.

[72] J. Eastep, D. Wingate, and A. Agarwal. Smart Data Structures: An Online Machine Learning Approach to Multicore Data Structures. In *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, pages 11–20, New York, NY, USA, 2011. ACM.

[73] R. Eigenmann, J. Hoeflinger, Z. Li, and D. A. Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing (LCPC), 1992*.

[74] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 2–15, New York, NY, USA, 2009. ACM.

[75] T. Elmas, A. Sezgin, S. Tasiran, and S. Qadeer. An annotation assistant for interactive debugging of programs with common synchronization idioms. In *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '09, pages 10:1–10:11, New York, NY, USA, 2009. ACM.

[76] N. En and N. Srensson. An Extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of

Lecture Notes in Computer Science, pages 333–336. Springer Berlin / Heidelberg, 2004.

[77] S. Feldman. A Fortran to C converter. In *ACM SIGPLAN Fortran Forum*, volume 9, pages 21–22. ACM, 1990.

[78] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.*, July 1987.

[79] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 79–90, New York, NY, USA, 2009. ACM.

[80] M. Frigo and S. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216 –231, 2005. `http://www.fftw.org/`.

[81] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[82] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor. Kremlin: rethinking and rebooting gprof for the multicore age. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 458–469, New York, NY, USA, 2011. ACM.

[83] P. Giannozzi, S. Baroni, N. Bonini, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, G. L. Chiarotti, M. Cococcioni, I. Dabo, A. Dal Corso, S. de Gironcoli, S. Fabris, G. Fratesi, R. Gebauer, U. Gerstmann, C. Gougoussis, A. Kokalj, M. Lazzeri,

L. Martin-Samos, N. Marzari, F. Mauri, R. Mazzarello, S. Paolini, A. Pasquarello, L. Paulatto, C. Sbraccia, S. Scandolo, G. Sclauzero, A. P. Seitsonen, A. Smogunov, P. Umari, and R. M. Wentzcovitch. QUANTUM ESPRESSO: A modular and open-source software project for quantum simulations of materials. *Journal of Physics: Condensed Matter*, 21(39):395502 (19pp), 2009. `http://www.quantum-espresso.org`.

[84] L. Gil, P. F. Flores, and L. M. Silveira. PMSat: A Parallel Version of MiniSAT. *JSAT*, 6(1-3):71–98, 2009.

[85] S. Graham, M. Snir, and C. Patterson. *Getting up to speed: The future of supercomputing*. National Academy Press, 2005.

[86] W. Gropp, E. Lusk, and A. Skjellum. Using MPI-Portable Parallel Programming with the Message-Passing Interface. *Sci. Program.*, 5:275–276, August 1996.

[87] D. Grossman. Type-safe multithreading in Cyclone. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '03, pages 13–25, New York, NY, USA, 2003. ACM.

[88] J. Gummaraju, J. Coburn, Y. Turner, and M. Rosenblum. Streamware: programming general-purpose multicore processors using streams. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 297–307, New York, NY, USA, 2008. ACM.

[89] H. Gunnar. Bobcat, 2011. `http://github.com/Bobcat/bobcat`.

[90] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2001.

[91] Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: A Parallel SAT Solver. *JSAT*, 6(4):245–262, 2009.

[92] H. Han and C. Tseng. Improving compiler and run-time support for adaptive irregular codes. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 393–400. IEEE, 1998.

[93] R. Haney, T. Meuse, J. Kepner, and J. Lebak. The HPEC challenge benchmark suite. In *HPEC 2005 Workshop*, 2005.

[94] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson. How do scientists develop and use scientific software? *Software Engineering for Computational Science and Engineering, ICSE Workshop on*, 0:1–8, 2009.

[95] T. Harris and S. Singh. Feedback Directed Implicit Parallelism. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2007.

[96] E. Hehner. *A practical theory of programming*. Springer, 1993.

[97] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 2006.

[98] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proc. of PPoPP*, 2008.

[99] O. Hernandez, C. Liao, and B. Chapman. Dragon: A static and dynamic tool for OpenMP. *Shared Memory Parallel Programming with Open MP*, pages 53–66, 2005.

[100] S. Hiranandani, K. Kennedy, C. Tseng, and S. Warren. The D editor: A new interactive parallel programming tool. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 733–742. ACM, 1994.

[101] K. J. Hoffman, H. Metzger, and P. Eugster. Ribbons: a partially shared memory programming model. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 289–306, New York, NY, USA, 2011. ACM.

[102] C. Huttenhower, M. Schroeder, M. Chikina, and O. Troyanskaya. The Sleipnir library for computational functional genomics. *Bioinformatics*, 24(13):1559–1561, 2008.

[103] C. Huttenhower, M. Schroeder, M. D. Chikina, and O. G. Troyanskaya. The Sleipnir library for computational functional genomics. *Bioinformatics*, 24(13):1559–1561, July 2008. `http://huttenhower.sph.harvard.edu/sleipnir/`.

[104] W.-m. Hwu, S. Ryoo, S.-Z. Ueng, J. Kelm, I. Gelado, S. Stone, R. Kidd, S. Baghsorkhi, A. Mahesri, S. Tsao, N. Navarro, S. Lumetta, M. Frank, and S. Patel. Implicitly Parallel Programming Models for Thousand-Core Microprocessors. In *Proceedings of the 44th annual Design Automation Conference (DAC)*, 2007.

[105] M. Isard and A. Birrell. Automatic mutual exclusion. In *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–6. USENIX Association, 2007.

[106] J. Jenista, Y. Eom, and B. Demsky. OoOJava: an out-of-order approach to parallel programming. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, pages 11–11. USENIX Association, 2010.

[107] D. Jeon, S. Garcia, C. Louie, and M. B. Taylor. Kismet: parallel speedup estimates for serial programs. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 519–536, New York, NY, USA, 2011. ACM.

[108] T. Joachims. Making large-Scale SVM Learning Practical. *Advances in Kernel Methods Support Vector Learning*, pages 169–184, 1999.

[109] N. Johnson, H. Kim, P. Prabhu, A. Zaks, and D. August. Speculative separation for privatization and reductions. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 359–370. ACM, 2012.

[110] C. Jones, P. O'Hearn, and J. Woodcock. Verified software: A grand challenge. *Computer*, 39(4):93–95, 2006.

[111] E. Jones, T. Oliphant, and P. Peterson. SciPy: Open source scientific tools for Python. 2001. `http://www.scipy.org/`.

[112] M. P. Jones and P. Hudak. Implicit and Explicit Parallel Programming in Haskell. Technical report, 1993.

[113] M. Julliard. Dynare: A Program for the Resolution and Simulation of Dynamic Models with Forward Looking Variables Through The Use of Relaxation Algorithm. *CEPREMAP*, 2005. `http://www.dynare.org/`.

[114] L. V. Kale and S. Krishnan. CHARM++: a portable concurrent object oriented system based on C++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[115] W. J. Kaufmann and L. L. Smarr. *Supercomputing and the Transformation of Science*. W. H. Freeman & Co., New York, NY, USA, 1992.

[116] M. Kawaguchi, P. Rondon, A. Bakst, and R. Jhala. Deterministic parallelism via liquid effects. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 45–54, New York, NY, USA, 2012. ACM.

[117] C. Ke, L. Liu, C. Zhang, T. Bai, B. Jacobs, and C. Ding. Safe parallel programming using dynamic dependence hints. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 243–258, New York, NY, USA, 2011. ACM.

[118] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: a Dependence-based Approach*. Morgan Kaufmann Publishers Inc., 2002.

[119] K. Kennedy, K. McKinley, and C. Tseng. Interactive parallel programming using the ParaScope Editor. *Parallel and Distributed Systems, IEEE Transactions on*, 2(3):329–341, 1991.

[120] D. Kim and M. C. Rinard. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 528–541, New York, NY, USA, 2011. ACM.

[121] G. Kollias, A. Grama, and Z. Li. Asynchronous Iterative Algorithms. In *Encyclopedia of Parallel Computing*, pages 87–95. 2011.

[122] D. Komatitsch, J. Labarta, and D. Michéa. A simulation of seismic wave propagation at high resolution in the inner core of the Earth on 2166 processors of MareNostrum. *High Performance Computing for Computational Science-VECPAR 2008*, pages 364–377, 2008. `http://www.geodynamics.org/cig/software/specfem3d`.

[123] D. Komatitsch, S. Tsuboi, C. Ji, and J. Tromp. A 14.6 billion degrees of freedom, 5 teraflops, 2.5 terabyte earthquake simulation on the Earth Simulator. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 4–, New York, NY, USA, 2003. ACM.

[124] M. Koshimura, T. Zhang, H. Fujita, and R. Hasegawa. QMaxSAT: A Partial Max-SAT Solver system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 8:95–100, 2012.

[125] E. Koskinen, M. Parkinson, and M. Herlihy. Coarse-Grained Transactions. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2010*.

[126] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. How much parallelism is there in irregular applications? In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 3–14, New York, NY, USA, 2009. ACM.

[127] M. Kulkarni, D. Nguyen, D. Prountzos, X. Sui, and K. Pingali. Exploiting the commutativity lattice. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 542–555, New York, NY, USA, 2011. ACM.

[128] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 211–222, New York, NY, USA, 2007. ACM.

[129] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[130] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of 2nd International Symposium on Code Generation and Optimization (CGO)*, 2004.

[131] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 227–242, New York, NY, USA, 2009. ACM.

[132] R. Leino, P. Müller, and J. Smans. Deadlock-free Channels and Locks. In *Proceedings of the 19th European Symposium on Programming (ESOP)*, 2010.

[133] M. Li, R. Sasanka, S. Adve, Y. Chen, and E. Debes. The ALPBench benchmark suite for complex multimedia applications. In *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*, pages 34–45. IEEE, 2005.

[134] X. Liang, D. P. Lettennmaier, E. F. Wood, and S. J. Burges. A simple hydrologically based model of land surface water and energy fluxes for general circulation models. 99:14415–14428, July 1994. http://www.hydro.washington.edu/Lettenmaier/Models/VIC/.

[135] S.-W. Liao, A. Diwan, R. P. Bosch, Jr., A. Ghuloum, and M. S. Lam. SUIF Explorer: an interactive and interprocedural parallelizer. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '99, pages 37–48, New York, NY, USA, 1999. ACM.

[136] Liberty Research Group. The Parallelization Project, 2012. http://liberty.princeton.edu/Projects/AutoPar/.

[137] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 287–296, New York, NY, USA, 2008. ACM.

[138] L. Liu and Z. Li. Improving parallelism and locality with asynchronous algorithms. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 213–222, New York, NY, USA, 2010. ACM.

[139] R. Lublinerman, S. Chaudhuri, and P. Cerny. Parallel programming with object assemblies. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 61–80, New York, NY, USA, 2009. ACM.

[140] S. Luke. Essentials of Metaheuristics, 2010.

[141] J. F. Martínez and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS X, pages 18–29, New York, NY, USA, 2002. ACM.

[142] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

[143] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 166–176, New York, NY, USA, 2009. ACM.

[144] G. Memik, W. H. Mangione-Smith, and W. Hu. NetBench: a benchmarking suite for network processors. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2001.

[145] Z. Merali. Why scientific programming does not compute. *Nature News*, 467, 2010.

[146] R. Meyers and Z. Li. ASYNC Loop Constructs for Relaxed Synchronization. In *Languages and Compilers for Parallel Computing*, pages 292–303. Springer-Verlag, Berlin, Heidelberg, 2008.

[147] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IEEE International Symposium on Workload Characterization (IISWC)*, September 2008.

[148] C. Moler. *Numerical computing with MATLAB*. Society for Industrial Mathematics, 2004.

[149] G. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82 –85, Jan 1998.

[150] The message passing interface (MPI) standard. http://www-unix.mcs.anl.gov/mpi/.

[151] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. MineBench: A Benchmark Suite for Data Mining Workloads. In *IEEE International Symposium on Workload Characterization (IIWSC)*, 2006.

[152] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for C/C++. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 195–212, New York, NY, USA, 2008. ACM.

[153] A. Z. Nick P. Johnson, Taewook Oh and D. I. August. Fast Condensation of the Program Dependence Graph. In *Proceedings of the 34rd ACM SIGPLAN Conference on Programming Language Design and Implementation (to appear)*, PLDI, New York, NY, USA, 2013. ACM.

[154] R. Nikhil. *Implicit parallel programming in pH*. Morgan Kaufmann, 2001.

[155] C. Oancea, A. Mycroft, and T. Harris. A lightweight in-place implementation for software thread-level speculation. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 223–232, 2009.

[156] T. Oliphant. *A Guide to NumPy*, volume 1. Trelgol Publishing, 2006. `http://numpy.scipy.org/`.

[157] G. Ottoni. *Global Instruction Scheduling for Multi-Threaded Architectures*. PhD thesis, 2008.

[158] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *In Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture*, pages 105–118. IEEE Computer Society, 2005.

[159] C. Pancake and C. Cook. What users need in parallel tool support: Survey results and analysis. In *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, pages 40–47. IEEE, 2002.

[160] I. Park, M. Voss, S. Kim, and R. Eigenmann. Parallel programming environment for OpenMP. *Scientific Programming*, 9(2-3):143–161, 2001.

[161] H. Payer, H. Roeck, C. M. Kirsch, and A. Sokolova. Scalability versus semantics of concurrent FIFO queues. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '11, pages 331–332, New York, NY, USA, 2011. ACM.

[162] C. Pheatt. Intel threading building blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, Apr. 2008.

[163] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *J. Comput. Phys.*, 117:1–19, March 1995.

[164] M. Plummer. JAGS: Just Another Gibbs Sampler. `http://www-ice.iarc.fr/~martyn/software/jags/`, 2011.

[165] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August. Commutative Set: A Language Extension for Implicit Parallel Programming. In *Proceedings of the 2011 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, New York, NY, USA, 2011. ACM.

[166] P. Prabhu, G. Ramalingam, and K. Vaswani. Safe programmable speculative parallelism. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 50–61, New York, NY, USA, 2010. ACM.

[167] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical recipes in Fortran: the art of scientific computing*. Cambridge university press, 1993.

[168] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[169] R. Rabbah, I. Bratt, K. Asanovic, and A. Agarwal. Versatility and Versabench: A new metric and a benchmark suite for flexible architectures. 2004.

[170] T. Ralphs and M. Güzelsoy. The SYMPHONY callable library for mixed integer programming. *The next wave in computing, optimization, and decision technologies*, pages 61–76, 2005.

[171] A. Raman, A. Zaks, J. W. Lee, and D. I. August. Parcae: A system for flexible parallel execution. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 133–144, New York, NY, USA, 2012. ACM.

[172] E. Raman, G. Ottoni, A. Raman, M. Bridges, and D. I. August. Parallel-Stage Decoupled Software Pipelining. In *Proceedings of the 2008 International Symposium on Code Generation and Optimization*, April 2008.

[173] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-Stage Decoupled Software Pipelining. In *Proceedings of the 6th annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2008.

[174] E. Raman, N. Vachharajani, R. Rangan, and D. I. August. Spice: speculative parallel iteration chunk execution. In *CGO '08: Proceedings of the 2008 International Symposium on Code Generation and Optimization*, pages 175–184, New York, NY, USA, 2008. ACM.

[175] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13 –24, feb. 2007.

[176] L. Rauchwerger, F. Arzu, and K. Ouchi. Standard Templates Adaptive Parallel Library (STAPL). In *LCR*, pages 402–409, 1998.

[177] L. Rauchwerger and D. A. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Trans. Parallel Distrib. Syst.*, 10:160–180, February 1999.

[178] J. Reppy. Concurrent ML: Design, application and semantics. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 165–198. Springer, 1993.

[179] M. C. Rinard. *The design, implementation and evaluation of Jade, a portable, implicitly parallel programming language*. PhD thesis, 1994.

[180] M. C. Rinard and P. Diniz. Commutativity Analysis: A New Analysis Framework for Parallelizing Compilers. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI)*, 1996.

[181] M. C. Rinard and P. C. Diniz. Eliminating synchronization bottlenecks using adaptive replication. *ACM Trans. Program. Lang. Syst.*, 25(3):316–359, May 2003.

[182] A. Sameh, G. Cybenko, M. Kalos, K. Neves, J. Rice, D. Sorensen, and F. Sullivan. Computational Science and Engineering. *ACM Comput. Surv.*, 28:810–817, December 1996.

[183] S. Sato and H. Iwasaki. Automatic parallelization via matrix multiplication. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 470–479, New York, NY, USA, 2011. ACM.

[184] P. Selinger. potrace: Transforming bitmaps into vector graphics. `http://potrace.sourceforge.net`.

[185] S. Sharma, R. Ponnusamy, B. Moon, Y. Hwang, R. Das, and J. Saltz. Run-time and compile-time support for adaptive irregular problems. In *Supercomputing'94. Proceedings*, pages 97–106. IEEE, 1994.

[186] O. Sinnen, R. Long, and Q. Tran. Aiding parallel programming with on-the-fly dependence visualisation. In *Parallel and Distributed Computing, Applications and Technologies, 2009 International Conference on*, pages 475–481. IEEE, 2009.

[187] D. Slate. A chess program that uses transposition table to learn from experience. *International Computer Chess Association Journal*, 10(2):59–71, 1987.

[188] K. Smith, B. Appelbe, and K. Stirewalt. Incremental dependence analysis for interactive parallelization. In *Proceedings of the 4th international conference on Supercomputing*, ICS '90, pages 330–341, New York, NY, USA, 1990. ACM.

[189] Standard Performance Evaluation Corporation (SPEC). http://www.spec.org.

[190] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. Streamflex: high-throughput stream programming in java. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 211–228, New York, NY, USA, 2007. ACM.

[191] Stanford Compiler Group. SUIF: A parallelizing and optimizing research compiler. Technical Report CSL-TR-94-620, Stanford University, Computer Systems Laboratory, May 1994.

[192] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 1–12, June 2000.

[193] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 2–13, February 1998.

[194] D. E. Stevenson. Science, Computational Science, and Computer Science: At a Crossroads. In *Proceedings of the 1993 ACM conference on Computer science*, CSC '93, pages 7–14, New York, NY, USA, 1993. ACM.

[195] J. Stone, T. Gardiner, and P. Teuben. Athena MHD Code Project. `http://trac.princeton.edu/Athena/`.

[196] M. Süß and C. Leopold. Implementing Irregular Parallel Algorithms with OpenMP. *Euro-Par 2006 Parallel Processing*, pages 635–644, 2006.

[197] V. Tabatabaee, A. Tiwari, and J. Hollingsworth. Parallel Parameter Tuning for Applications with Performance Variability. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, page 57, nov. 2005.

[198] E. Talbi. *Parallel combinatorial optimization*, volume 58. Wiley-Blackwell, 2006.

[199] G. Teodoro and A. Sussman. AARTS: low overhead online adaptive auto-tuning. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 1–11, New York, NY, USA, 2011. ACM.

[200] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 12th International Conference on Compiler Construction*, 2002.

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

[202] O. Tripp, G. Yorsh, J. Field, and M. Sagiv. HAWKEYE: effective discovery of dataflow impediments to parallelization. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 207–224, New York, NY, USA, 2011. ACM.

[203] A. Udupa, K. Rajan, and W. Thies. ALTER: exploiting breakable dependences for parallelization. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 480–491, New York, NY, USA, 2011. ACM.

[204] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative Decoupled Software Pipelining. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 49–59, Washington, DC, USA, 2007. IEEE Computer Society.

[205] H. Vandierendonck, S. Rul, and K. De Bosschere. The Paralax Infrastructure: Automatic Parallelization With a Helping Hand. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[206] H. Vandierendonck, S. Rul, and K. De Bosschere. The Paralax infrastructure: automatic parallelization with a helping hand. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 389–400, New York, NY, USA, 2010. ACM.

[207] S. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. Taylor. SD-VBS: The San Diego vision benchmark suite. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 55–64. IEEE, 2009.

[208] C. von Praun, L. Ceze, and C. Caşcaval. Implicit Parallelism with Ordered Transactions. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2007.

[209] G. Wang, P. Cook, et al. ChucK: A concurrent, on-the-fly audio programming language. In *Proceedings of International Computer Music Conference*, pages 219–226. Citeseer, 2003.

[210] Z. Wang and M. F. O'Boyle. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 307–318, New York, NY, USA, 2010. ACM.

[211] W. E. Weihl. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Trans. Comput.*, 37, December 1988.

[212] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[213] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 439–453, New York, NY, USA, 2005. ACM.

[214] N. Wirth. A plea for lean software. *Computer*, 28(2):64–68, 1995.

[215] Y. Wong, T. Dubrownik, W. Tang, W. Tan, R. Duan, R. Goh, S.-h. Kuo, S. Turner, and W.-F. Wong. Tulipse: A Visualization Framework for User-Guided Parallelization. In C. Kaklamanis, T. Papatheodorou, and P. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 4–15. Springer Berlin Heidelberg, 2012.

[216] P. Wu and D. A. Padua. Beyond Arrays - A Container-Centric Approach for Parallelization of Real-World Symbolic Applications. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing (LCPC), 1999*, 1999.

[217] Q. Wu, A. Pyatakov, A. N. Spiridonov, E. Raman, D. W. Clark, and D. I. August. Exposing Memory Access Regularities Using Object-Relative Memory Profiling. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2004.

[218] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. S. Lee. Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2008.

[219] W. Zhang. Phase Transitions and Backbones of 3-SAT and Maximum 3-SAT. In T. Walsh, editor, *Principles and Practice of Constraint Programming CP 2001*, volume 2239 of *Lecture Notes in Computer Science*, pages 153–167. Springer Berlin / Heidelberg, 2001.

[220] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering Hidden Loop Level Parallelism in Sequential Applications. In *Proceedings of 14th International Conference on High-Performance Computer Architecture (HPCA)*, February 2008.