Building a Foundation for the Future of Software Practices within the Multi-Core Domain

by

Celina Berg
B.Sc., University of Victoria, 2005
M.Sc., University of Victoria, 2006

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

Building a Foundation for the Future of Software Practices within the Multi-Core Domain

by

Celina Berg
B.Sc., University of Victoria, 2005
M.Sc., University of Victoria, 2006

Supervisory Committee

Dr. M.Y. Coady, Supervisor
(Department of Computer Science)

Dr. H.A. Müller, Departmental Member
(Department of Computer Science)

Dr. A. Thomo, Departmental Member
(Department of Computer Science)

Dr. A. Gulliver, Outside Member
(Department of Electrical and Computer Engineering)

Supervisory Committee

Dr. M.Y. Coady, Supervisor
(Department of Computer Science)

Dr. H.A. Müller, Departmental Member
(Department of Computer Science)

Dr. A. Thomo, Departmental Member
(Department of Computer Science)

Dr. A. Gulliver, Outside Member
(Department of Electrical and Computer Engineering)

ABSTRACT

Multi-core programming presents developers with a dramatic paradigm shift. Where the conceptual models of sequential programming largely supported the decoupling of source from underlying architecture, it is now unwise to develop new patterns, abstractions and parallel software in complete isolation from issues of modern hardware utilization. Challenging issues historically associated with complex systems code are now compounded within the parallel domain. These issues are manifested at all stages of software development including design, development, testing and maintenance. Programmers currently lack the essential tools to even partially automate reasoning techniques, resource utilization and system configuration management. Current trial and error strategies lack a systematic approach that will scale to growing multi-core and multi-processor environments. In fact, current algorithm and data layout conceptual models applied to design, implementation and pedagogy often conflict with effective parallelization strategies. This disertation calls for a rethinking, rebuilding and retooling of conceptual models, taking into account opportunities to introduce parallelism for multi-core architectures from the ground up. In order to establish new conceptual models, we must first 1) identify inherent complexities in multi-core development, 2) establish support strategies to make handling them more

explicit and 3) evaluate the impact of these strategies in terms of proposed software development practices and tool support.

Contents

List of Tables

List of Figures

Acknowledgements

First and foremost I would like to thank my supervisor and friend, Dr. Yvonne Coady. Always answering a question with another question, you guided my development as a researcher and provided me with endless opportunities to expand my horizons and grow as a person. In the best of times and the worst of times, your boundless energy and enthusiasm wrapped me like a wetsuit and kept me afloat. Thanks Coach!

To the members of the MOD(ularity) Squad, thank you for providing insightful feedback and an always amusing working environment. Jen, Chris and Onat – I could not have found better colleagues and friends to have begun and finished this journey with.

Thank you to my committee members, Dr. Aaron Gulliver, Dr. Hausi A. Müller and Dr. Alex Thomo, for holding me to a high standard and providing feedback that helped to construct and polish this dissertation. Additionally, thank you to Dr. Jeff Gray for agreeing to be the external examiner for this dissertation. Your attention to detail, stimulating questions and thoughtful observations helped to shape the final version of this dissertation.

To my parents and my siblings, thank you for your undying support, encouragement and pride. LeeAnne, not only did you always tell me I could do anything I wanted to do – you made me believe it.

Sean and Michele, and my extended Spilsbury family that have become such a fundamental part of my life – I know I could not have completed this without your support.

Haley and Georgia, you have been by my side for every success and every bump in the road. From beginning to end, you were a part of this journey. This achievement is just as much yours as it is mine.

Brad, although you have only recently joined this journey, you have made it complete. Everything is so much sweeter when shared with you.

Dedication

To Haley, the Sunshine that brightens my day.
and
To Georgia, the Buckaroo that lightens my heart.

Chapter 1

Introduction

In *The Mythical Man Month* Brooks discusses the impact of good software development practices and the importance of having a prominent design phase within the development process (Brooks, 1975). Establishing a counter intuitive law of "adding manpower to a late software project makes it later", Brooks was confident from experience with the evolution of OS360, that adherence to better software practices would more likely keep a project on schedule than bringing in more developers. In order to reap the benefits of extra manpower, coordination across those developers is necessary. Brooks' OS360 case study showed that effort would be better spent coordinating across existing developers rather than adding more manpower. Since then, fields of study have been forged to address these issues—research in processes, tools and formal methods share a common goal to provide sound practices, combined with support mechanisms to improve the productivity of programmers and the quality of their software.

We are moving into a new age of programming. In 1965, Moore's law (Moore, 1965) predicted that the number of transistors on a single chip would approximately double every 18 months, doubling performance benefits for free. This law of exponential growth was lengthened to a period of every two years, holding in a single processor environment until hardware reached the limits in terms of how close transistors can be placed on a single chip (Intel Corporation, 2005). In keeping with Moore's law, the placement of multiple cores on a chip are facilitating the growth of the number of transistors on a chip, but the performance benefits for developers working with these multi-core architectures is no longer free. They now must understand how to make use of multiple cores in order to reap the performance benefits.

The differing perspectives put forth by Grace Hopper and Seymour Cray illustrate

the sense of tradeoffs between the possible performance gains of increasing the number of processing elements versus the cost of organizing those processing elements to complete a task:

> "In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers."
> ~ *Rear Admiral Grace Hopper* (Lewis, 2010)

> "What would you rather have to plow a field—two strong oxen or 1,024 chickens?"
> ~ *Seymour Cray* (Star Quotes, 2010)

Hopper's proposal was realized in the past with distributed and parallel computing and now this subsequent cap on single-core processor speeds has put us in a position where we are essentially faced with fine-grain composition of hardware resources to keep up with Moore's law. But as Cray alludes to, there is difficulty and inherent overhead associated with the use of multiple resources. Theoretically, doubling the computational resources should double the performance of an application but this is not the case when this management overhead is taken into account. We have reached a limit to the processing capacity for a single chip, but the organization of computation across multiple compute units introduces overhead not only in terms of performance, but perhaps more importantly, in terms of a programmer's cognitive load. Essentially, this same issue ironically lies at the heart of Brooks' Law, where coordination or management costs start to dominate throughput in real-world, software development scenarios.

Multi-processor and multi-core architectures are becoming mainstream and even Graphical Processing Units (GPUs) are being leveraged for general computation (Harris, 2005), but associated benefits of these hardware advances cannot be achieved if applications are not able to make use of processing elements available to them. The introduction of NVIDIA's Tesla supercomputer, housing an astonishing 960 cores for less than $10,000, demonstrates the reality of our situation (NVIDIA Corporation, 2008). Current trajectories suggest that future hardware platforms will house thousands of cores, as those that contain millions are already actively undergoing testing, such as IBM's project KittyHawk (Appavoo *et al.*, 2008). While hardware is advancing beneath us, the question as to how to make full use of these resources is

described by John Hennessy, president of Stanford University, as "the biggest problem Computer Science has ever faced" (O'Hanlon, 2006).

Parallelism introduces critical issues such as resource utilization and contention which had largely been factored out of mainstream development practices for high-level applications executing in a sequential environment. Though parallelism itself is not a new challenge, the current state of flux for applications and the degree to which they need to be transformed is relatively new and somewhat alarming.

In his article, *The Free Lunch is Over*, Herb Sutter describes concurrency as "the next major revolution in how we write software", but at the same time forewarns about the complexities of development in this domain (Sutter, 2005). Sutter goes on to argue that the programming model for parallel applications is much more difficult to master than that of a sequential model, giving examples of unexpected program behaviour and overhead of *safe* programming resulting in applications that are no faster than they would be in a single processor environment. The complexity that introduces insidious, unexpected program behaviour and performance overhead stems from the need to coordinate at multiple levels, including organization of tasks, access to shared data and resource utilization. This thesis shows how organization or *coordination* of tasks, data and resources manifests itself as one of the biggest and most challenging parts of programming for parallel architectures.

The daunting task of efficient programming for highly parallel systems is currently receiving much attention from several perspectives within computer science. In *The Landscape of Parallel Computing Research* (Asanovic *et al.*, 2006) from UC Berkeley, the authors draw on lessons learned about parallelism across all areas of computing to recommend a generalized, pattern-based approach to designing parallel applications in order to allow them to scale to thousands of cores.

These new multi-processor architectures requires new approaches to achieve performance gains, forcing developers to think hard about the dynamic impact of static changes to software. This new age offers an opportunity for researchers to rethink programming models, system software, and hardware architectures from the ground up driven by performance goals. Amdahl's law (Amdahl, 1967), simply put, can be used to find the maximum expected speedup for an overall system when only part of the system can be parallelized.

$$\frac{1}{(1-P)+\frac{P}{N}} \tag{1.1}$$

Equation 1.1 is the mathematical representation of a maximum speed up according to Amdahl's Law, where P is the amount of code that can be parallelized and N is the number of processing units. As N increases the value of $\frac{P}{N}$ becomes negligible and the equation resolves to $\frac{1}{(1-P)}$, leaving the speedup to be determined by how much of the program must remain sequential. Ultimately, the maximum speed of an application is limited by the speed of the sum of the sequential parts of the application. What this raw calculation does not take into account is the overhead incurred by the coordination required to parallelize the computation across processing units.

The question as to how to programmatically make full use of these multi-core resources has become the latest challenge. Existing software engineering practices do not take into account the subtle, yet critical, issues of utilization and contention. Current manual approaches to parallelism are largely based on trial and error, brute force and *ad hoc* approaches and lack the structure that can be provided by explicit software engineering practices.

1.1 The Thesis

The thesis of this research is that software development practices and integrated environments can better support parallel development with a model that structures the acquisition and tracking of tangible artefacts key to the parallel development cycle. Systematic processes and parallel-specific conceptual models can provide support for analysis and comparison within and across these artefact from an implementation level to a level of abstraction that supports reasoning about these artefacts as a knowledge base.

This chapter provides a survey of the domain in terms of existing infrastructure, applications and language support (Section 1.2). This background is intended to provide understanding of the evolution of computer architectures and provide a sense of the issues developers are faced with now and in the years to come. A survey of general related work in the area of support strategies for parallel application development establishes context for the solution space of this thesis (Section 1.3). This survey is intended to provide an understanding of how development practices have progressed over the years, advancing through lessons learned in trying to build quality software. Finally, an overview of the organization and contributions of the thesis is provided (Section 1.4).

1.2 Background and Motivation

This dissertation provides strategies to support developers through structured practices and useful tools that provide information relevant to the parallel programming process. An overview of architecture evolution provides a view of this challenging domain and motivates today's need to support parallel development (Section 1.2.1). Several scientific communities have demonstrated the ability to leverage parallel architectures for repetitive algorithms and mass computation. A survey of existing applications that have shown substantial performance benefits from parallelization (Section 1.2.2) is followed by an introduction to existing parallel linguistic support (Section 1.2.3) and a discussion of tradeoffs between them (Section 1.2.4). While linguistic support for multi-core development is in its inception, understanding how these mechanisms compare in terms of integration into existing languages, programming overhead and programmer control is key to moving forward in language development for the parallel domain.

1.2.1 Architectures

The architectural model introduced by von Neumann in 1945 (von Neumann, 1988) and realized in the first general-purpose electronic computer, ENIAC (Electronic Numerical Integrator And Computer) (Goldstine and Goldstine, 1996), only considered a single processor handling sequential processing. By 1976, with the Cray-1 (Russell, 1978), the age of the supercomputer reigned. Multi-processing machines entered the scene in 1982, leveraging parallelization in the Cray-2 and the Cray X-MP (Simmons and Wasserman, 1988).

Commodity personal computers are now quickly doubling the number of cores on a chip with dual and quad-core processors becoming commonplace with more than one of these multi-core processors on a single machine. This growth shows no signs of slowing, with Intel projecting 32 cores by 2010 (Gruener, 2006), but in fact unveiling 48 cores in 2009 (Bradley, 2009). The Cell Project (IBM, 2008) has introduced a power-efficient, cost-effective, high-performance, heterogeneous multi-core architecture to both commodity game consoles (Playstation 3 (PS3)) and server architectures (Cell Blades) (IBM, 2008).

For decades, computer research has been trying to break out of the sequential programming model to make full use of processing resources of parallel and distributed systems. Even in 1978, Backus posed the question *Can Programming Be*

Liberated from the von Neumann Style?, proposing a functional style programming solution (Backus, 1978). Perhaps developers have lacked the motivation to change until now, faced with multi-processor, multi-core, many-core and multi-threaded architectures. These architectures are able to execute multiple instructions in parallel on multiple processing units with possibly heterogeneous processing capabilities and resources.

The wide variety of existing parallel architectures have been designed to target specific sets of frequently occurring problems in the parallel domain. *Multiple Instruction stream, Multiple Data (MIMD)* hardware targets task-level parallelism—that is, the execution of independent tasks on multiple compute units. An example of the application of MIMD architecture was used in the Cray multi-processing machines (Simmons and Wasserman, 1988). MIMD processors perform independent, asynchronous computation on independent data held in memory on either shared or distributed. On the other hand, data-level parallelism, distributing data across multiple compute units and the simultaneous execution of the same task can be achieved with a *Single Instruction, Multiple Data (SIMD)* hardware model. SIMD has been applied in new high-performance GPUs, such as Intel's Larrabee (Seiler *et al.*, 2008). SIMD, first used in supercomputers, is now applied in personal computers and commodity devices and uses a vector processor approach.

In a vector processor, the same computation is performed on a set of values in a vector or array style data structure that aligns with the underlying hardware. This contrasts with a *scalar* approach, where the processor performs computation on a single value at a time. *Vectorization* is supported by existing languages through the use of *intrinsics*, which provide functionality optimized by the compiler. That is, a function call introduced by a programmer will be substituted by a set of instructions that map directly to a specific architecture. In the case of vector intrinsics, the compiler maps to vector style hardware.

GPUs are specialized microprocessors originally intended to accelerate graphics for use in game consoles, personal computers and hand held devices. The highly-parallel and low-cost nature of graphics hardware is now making this architecture desirable for programs requiring the execution of identical tasks on large data sets (Harris, 2005). In fact, the cost-effective nature of GPUs is making software a viable solution over an otherwise application-specific hardware solution. Currently, the Herzberg Institute of Astrophysics (National Research Council Canada (NRCC), 2010) is weighing this dilemma, considering a GPU solution for the adaptive optics software for a 30

meter telescope (Hovey *et al.*, 2010) as opposed to the proposed application-specific, customized hardware solution (Herriot *et al.*, 2010).

1.2.2 Application Domain

Many applications such as scientific computing, model checking, and signal processing, are described as being *embarrassingly parallelizeable* (Foster, 1995). This term refers to a problem for which the work can be performed on a data set in separate autonomous units that execute in isolation aligning well with the data parallelism of the SIMD architecture of today's GPUs. The term itself suggests a certain simplicity to the solution, but the question is, what would have to dramatically change to further enable more applications to reap the benefits of parallelism in today's development environment?

One example from scientific computing that is already taking advantage of GPUs is that of DNA sequencing. The repetitive-style task of DNA sequencing on a massive data set has been shown to reap the benefits of parallel architectures. Substantial speedups by a factor of 10 to 80 times have been demonstrated with genetic programs on small data sets by leveraging linguistic support for GPU programming (Robilliard *et al.*, 2008).

Model checking employs a logic description of a program to automatically verify correctness properties of finite-state systems, a problem that ultimately reduces to a graph search. These verification searches can get so large that in a practical sense, time does allow them to complete. An increase in coverage, that is, a direct increase in the number of states, from 8GB to 64GB increases a week of computation to a month (Holzmann *et al.*, 2010). In fact, it is accepted that for large verification problems a time-bounded search for errors cannot be completed. Swarm (Holzmann *et al.*, 2008) is a script generator for the Spin model checker (Havelund *et al.*, 2008) that leverages parallel capabilities to increase the coverage for very large verification problems by about an order of magnitude. Swarm was developed by altering a divide-and-conquer style algorithm to take advantage of parallel architectures and distributed systems. This application provides a highly-configurable interface to define the number of processing units, memory limits and time limits. It is likely that this kind of interface would be amenable to many scientific programmers.

A similarly computationally intensive algorithm is the Three-Dimensional Symmetrical Condensed Node Transmission Line Matrix (3D-SCN TLM) to calculate

electromagnetic fields involving substantial matrix multiplication. Leveraging GPUs for computation of a 3D-SCN TLM method showed a 120 times speedup over a commercially available 3D-SCN TLM solver (Rossi, 2010). This speedup not only required an efficient revamping of the algorithm to ensure memory locality was favourable for parallelization, but also the application of multiple aggressive optimization strategies that required intimate knowledge of the underlying architecture.

When working with these architectures, overhead generally stems from the need to move data around to parallelize the computation. This data movement overhead can dominate performance to the point where performance benefits of parallelization are lost, despite what Amdahls law might otherwise indicate. Costs are associated with data transfer, including transfer to and from CPU and GPU memory, and local memory on GPU cores (NVIDIA Corporation, 2009; Kirk and Hwu, 2010; Ryoo *et al.*, 2008). Thus, memory bandwidth is typically a key factor that needs to be optimized according to hardware specifications. With hardware that performs such large amounts of computation so quickly, the performance can be seriously impacted by the speed at which data can be transferred for computation and can become a performance bottleneck. Today's theoretical bandwidth rates for PCIe x16 cards are 8 GB/s (PCI SISIG, 2010) but often these rates are not realized. For example, a contributor to the *Ompf* discussion board writes, "ATI OpenCL barely reaches the 1GB/s transfer rate... it is a long standing "bug" they have never fixed" (Ompf, 2010).

Bandwidth optimizations can have an order-of-magnitude impact on the performance of memory accesses, and further are tied to subtle issues such as the number of threads attempting to access memory concurrently. For example, NVIDIA GPUs use the notion of a *thread-block*, where each block can contain up to 512 threads (NVIDIA Corporation, 2009). Each core executes one thread-block at a time, and the entire grid of thread-blocks is scheduled across the cores of the GPU in a coordinated fashion. Figure 1.1 (Rossi, 2010) shows variation in transfer rates associated with memory bandwidth (GB/sec) between GPU global and local core memory as the number of threads per thread-block increases. This behaviour is associated with what is commonly known as an optimal *coalescing condition* (noted first at 96 threads per block in Figure 1.1) when the most effective bandwidth is achieved.

Though this condition is a small detail that is completely architecture centric, it serves as an example of the current reality facing programmers attempting to leverage GPUs for performance gains. A naïve design that does not incorporate these features

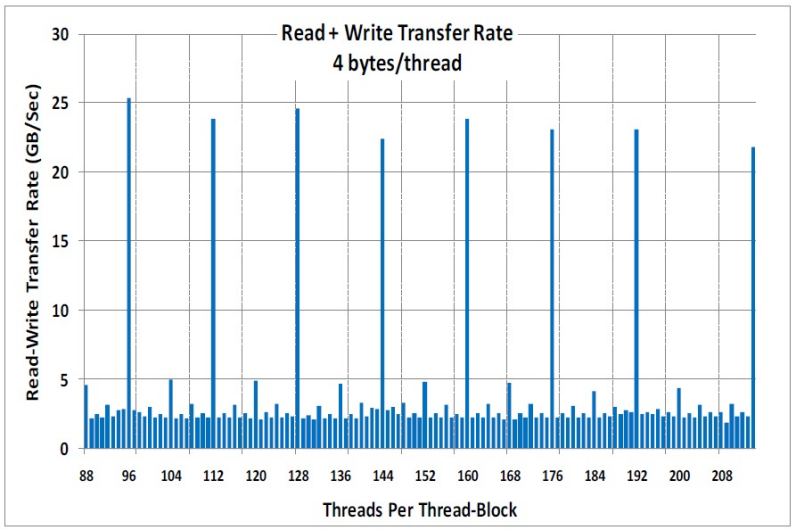

Figure 1.1: GPU Read-Write transfer rates between global memory and the multiprocessors when varying thread-block dimensions (Rossi, 2010).

may still achieve an order of magnitude speedup over the sequential version. However, the fact that these optimizations yield a second order of magnitude due to better resource utilization highlights the possibility that we may need to reconsider software engineering practices in general for performance sensitive applications running on these architectures.

1.2.3 Linguistic Support

The parallel domain is supported by a rapidly growing number of linguistic mechanisms, ranging from libraries to full languages with underlying compiler support. No one linguistic mechanism has been deemed the clear winner in this space. That is, different mechanisms have varying tradeoffs including usability, control of and correspondence to underlying architecture. This subsection gives an overview of some of these contributions from both industry and academia, concluding with preliminary studies that begin to compare and assess these linguistic constructs.

1.2.3.1 Libraries

Parallel support in the form of a library Application Programming Interface (API) allows for the augmentation of existing code bases through the introduction of explicit library calls, often providing a less invasive and adaptable solution than a complete

re-implementation. Dijkstra's original introduction of *semaphore* support was in the form of a library (Dijkstra, 1965). Thread libraries such as PThreads (Institute of Electrical and Electronics Engineers, 2004), provide C programmers with fine-grained control of synchronization of multi-threaded programs with semaphores and *mutexes*. When not applied with careful consideration of interactions, threading can introduce unexpected or incorrect behaviour in a program (Ousterhout, 1996).

For example, a *race condition* can occur when the execution of two processes or threads can result in different outcomes depending on the order in which they change or access shared state (Netzer and Miller, 1992). Similarly, a *deadlock* can occur if two or more processes are trying to access some shared set of resources but no progress can be made. In order for deadlock to occur, the resources must be used by the processes in mutual exclusion and cannot be preempted. Each process requires more than one resource, and acquires control of at least one of them and does not relinquish control of that resource until it gains access to the others it requires that are in turn being held by other processes. All processes are left waiting to acquire at least one resource being held by another process, forming a circular dependency and preventing progress by any of them (Zöbel, 1983).

These common issues associated with concurrency are best described in computer science as a property of systems: "several computations executing simultaneously or interleaved while potentially interacting with each other" (Tanenbaum, 2007). Concurrency can be introduced on one processing unit with an interleaving of work.

Message Passing Interface (MPI) (Snir and Otto, 1998) provides a level of abstraction similar to message driven execution (Gürsoy and Kale, 2004) giving developers access to parallelism through library calls. This API provides a means for initializing a set of processes that are mapped to computers or nodes in a distributed or parallel system and newer MPI versions that provide programming support for shared memory architectures. *Send* and *receive* messages facilitate communication between these processes in terms of data distribution and work to be completed. MPI provides a much higher level of abstraction for the developer than that of PThreads, which exposes many more details such as control over threads and access to shared memory.

OpenMP is a language-independent programming support for multi-platform, shared memory architectures that is somewhat transparent, leveraging compiler directives, library routines and environment variables. For example, developers can precede parallelizeable loops in a code base with `#pragma` statements. This approach provides some allowance for hand-tuning but the parallelization details are mainly

left to the underlying compiler. This approach lends itself to the transformation of existing sequential applications to parallelized versions and is particularly flexible and noninvasive in terms of source code modifications required.

RapidMind (Monteyne, 2008), a commercial library, provides a multi-core development platform in C++ (Stroustrup, 2000) through which applications automatically leverage multi-processor architectures. RapidMind provides the automatic division of data and computation across cores in a *Single Program, Multiple Data (SPMD)* stream processing model (Darema, 2001). SPMD does not impose a simultaneous execution of instructions and can be executed on general-purpose CPUs, unlike the SIMD model. Due to the proprietary nature of RapidMind, it is difficult to analyze or compare their approach relative to alternatives.

Though there are many other libraries that can be considered useful in this domain, there are also those that hold promise in terms of their cognitive support as they continue to mature in terms of their performance gains. The conceptual model of an atomic transaction in *Transactional Memory* (Herlihy and Moss, 1993) was integrated into both C (Harris and Fraser, 2003) and Java (Flanagan and Qadeer, 2003) as support libraries rather than formal language support. *Automatic Mutual Exclusion (AME)* (Isard and Birrell, 2007) forces all shared state to be protected by default unless explicitly specified otherwise. This *correctness-before-performance* approach shields developers from the underlying complexity yet provides the means to introduce fine-grained synchronization optimizations with simple language constructs. Each of these libraries provide support that shields a developer from the low-level details of parallelism, but each at its own cost.

1.2.3.2 Languages

Academic interest in making use of multiple resources for one job or application is not new. Argus (Liskov, 1988) and Emerald (Hutchinson, 1987) are both object-based languages specifically designed to support distributed computing. Argus, developed with banking systems in mind, provided synchronization through atomic transactions. While dynamic reconfiguration was supported by Argus, performance was dependent on programmers placing the computation and associated data on a single node. Emerald did not focus on fault tolerance, but looked to provide a simpler interface for distributed programming and to lower the cost of remote object invocation.

Following these languages for distributed computing, Parallax (Lewis and El-

Rewini, 1993) provides a graphical programming paradigm for distributed computing. This approach is based on a mainly static perspective, where a Parallax program is comprised of nodes (computation) and links (data flow). A graphical editor allows a developer to construct computation graphs that will ultimately be compiled and run. In this environment, a developer must specify properties to allow Parallax to automatically handle replication for fault tolerance.

Formal languages for modeling of parallel applications have also been developed to represent issues associated with developing parallel applications such as concurrency. Specifically, Milner developed a formal system to represent communication and concurrency found in systems dealing with interacting compute units (Milner, 1989). A formal set of semantics was developed in this work similar to early work by Hoare with Communication Sequential Processes (CSP) (Hoare, 1978). CSP also supports the description of concurrent interaction patterns. While categorized as a formal language, it has been used for the specification and verification of concurrent portions of systems in industry.

This formal system may be used both as a model in which to represent the behaviour of existing systems of computing agents or as a language in which to program desired systems. The notion of acceptance semantics is introduced and it is in terms of this that we give meaning to programs constructed in our framework.

Domain-specific languages (DSLs) (Fowler, 2010) provide compiler support tailored to a specific kind of application or architecture and in many cases, the compiler handles the low-level details of optimization. For example, NPClick provides linguistic support for network-specific task allocation (Plishker *et al.*, 2004). The domain-specific approach provides language mechanisms with a clearer expression of a solution to a specific problem than those of existing, mainstream languages. For example, with the complexity associated with programming for the Cell architecture, domain-specific compiler techniques have been developed to automate the generation of highly efficient code for the Cell architecture, optimizing for memory layout and a heterogenous processor model (Eichenberger *et al.*, 2005). StreamIT is another DSL targeting large streaming applications in which developers program a sequence of functions to be applied to each element in a stream of data. Specifically, the StreamIT compiler (Gordon *et al.*, 2002), is designed to handle the specific details of partitioning and scheduling of tasks in highly interactive architectures.

Erlang (Armstrong, 2007) is an open source, functional programming language developed by Ericsson Computer Science Laboratory (Ericsson Computer Science

Laboratory, 2010). Although it was first developed in the 80's, even then it provided linguistic support to simplify concurrent programming with process management support and message passing communication between either local or remote processes with no shared data or state, to avoid the need for locks. Erlang continues to evolve with computer architectures, now supporting shared memory, multi-processor architectures. This language provides garbage collection support, dynamic typing and code changes at runtime and eliminates the need for shared data through copying. While used by some large companies, Erlang has not yet become mainstream.

These optimizations can also be achieved manually using extensions to existing languages such as IBM's X10 (Murthy, 2008) project, which provides parallelization support within Java for cluster or grid computing. River (Arpaci-Dusseau *et al.*, 1999) provides a programming model in C++ to establish maximum performance across heterogeneous resources for I/O intensive applications. This *dataflow* style programming environment involves a set of concurrently executing processes that communicate similarly to message passing systems, by passing data via channels between processes. Each component within this type of application is a process with at least one input and one output. Dynamic load balancing across these components is handled through a distributed queue leveraging message passing for communication.

Similarly, Cilk, a C extension for multi-threading, supports dynamic load balancing through a *work-stealing scheduler*. The scheduler capitalizes on applications being broken into a sequence of threads which can spawn child threads that can be dynamically moved between processing units in a shared memory model (Randall, 1998). This multi-threaded approach has been shown to be better for expressing tree-like, recursive-style computations that conform to the same structure as the classic Fibonacci computation.

Given that parallel languages have the potential to substantially differentiate user experiences with software dealing with large data sets, industry is interested in developing linguistic support for a less complicated way to harness the potential of parallel processing. The key contenders such as Microsoft, Apple and Google are weighing into the challenge. The Open Quark Framework (Business Objects, 2007) from Business Objects allowed for the introduction of functional style programming constructs to be introduced into a Java program in the form of its own functional language, CAL (Business Objects, 2007). The functional programming paradigm, introduced originally with Lisp (McCarthy, 1978), is based on functions without side effects; that is, the execution of a function does not change the program state. This independence

makes functional languages much more conducive to concurrent programming. For example, `map` is a functional primitive which applies a defined function to all elements in a list in no particular order. This solution aligns closely with the definition of an embarrassingly parallel problem. Microsoft developed a similar extension to C# and Visual Basic called LINQ (language-integrated query) (Pialorsi and Russo, 2007) that allows for the introduction of similar constructs in the form of queries and transforms. Further, the PLINQ (parallel language-integrated query) (Duffy and Essey, 2007) extension leverages the embarrassingly parallelizeable nature of these functions and handles the distribution of data and computation across multiple processors or cores automatically for the .NET developer.

An Apple initiative, OpenCL (Apple, 2008), is a domain-specific framework that is currently being developed to support efficient programming of data and task parallelization across a pool of multiple processing units that can be a mix of CPUs and GPUs. This framework, being standardized by the Khronos group, looks to provide abstractions of underlying hardware specifics through language mechanisms. Developers must implement the basic unit of code called a *compute kernel*, which can be grouped to take advantage of data parallelization or, alternatively, leverage task parallelism. NVIDIA's CUDA (NVIDIA Corporation, 2010) framework provides a similar form of linguistic support but is architecture-specific and limited to programming of GPUs.

Intel's *Ct* (Intel Corporation, 2008a) language, also referred to as *C for throughput*, is implemented as a C++ extension and focuses on GPU architectures but is flexible to support programming for CPUs as well. This data parallel programming model is dynamically compiled, and is offered up to the programmer in the form of template-style types and operators for vector programming. In terms of the underlying implementation, Ct applies a fine-grained, dataflow threading model like River, in which computation is decomposed into a task dependence graph. The compiler handles the merging of similar tasks into coarser-grained tasks and even individual tasks are made data parallel. The Ct programming environment provides more control to experts through lower-level abstractions that expose task granularity, vector API intrinsics and optimizations. Interestingly, though both Ct and the C++ library support of RapidMind provide proprietary solutions in this problem space, they both leverage the lower-level support provided by OpenCL.

The Dryad (Isard *et al.*, 2007) programming model has also been integrated with C++, allowing developers to establish communication patterns and dynamically tune

distributed applications through syntactic support for building graph-based representations of communication and data flow. Google's Sawzall (Pike *et al.*, 2006) interpreted language exposes low-level infrastructure developed by Google for scheduling, data distribution and fault tolerance which leverages the concept of `map` from functional programming to create parallel *MapReduce* applications. Authors claim that in comparison to those written in C, Sawzall's MapReduce applications are simpler, making them both easier to write and understand.

Google provides in-house support for parallel programming with Sawzall and concurrency mechanisms to the mainstream programmer with the Go programming language (Google, 2009; Google Groups, 2010). Go is a statically typed compiled language and boasts support for a flexible and modular program construction with garbage collection and runtime reflection support which allows the modification of a program at runtime based on observations of the dynamic result of the application. A `goroutine` appears to be much like a function, but executes in parallel with other `goroutines` in the same address space, hiding the complexities of thread creation and management. A `channel` is an abstraction that combines communication with synchronization, and may be created as buffered or unbuffered. With a `channel`, one `goroutine` can be constructed to wait for another `goroutine`. For example, a receiver always blocks until there is data to receive. Likewise, if a `channel` is buffered, the sender will block if the buffer is full. In the unbuffered case, the sender blocks until the receiver has accepted the value. Using these building blocks, a key principle in Go is to avoid sharing memory. Instead, explicit communication is used—shared variables are passed through channels. At any point in time, only one `goroutine` has access to a shared value, making it easier to write code that is clear and correct.

Finally, in terms of system-level support, Android (Android, 2008), an open source operating system for hand-held devices, believes the answer is to maintain a Java-style API that developers are accustomed to. Android's operating system, middleware and key applications deal with power management and networking details in multi-core environments while providing a simple programming interface for customizations. Along these lines, virtual machine and compiler options are also starting to reflect the changes in hardware platforms. Though these options have long-term potential to shield application developers from low-level details, until their dynamic characteristics and tradeoffs are better understood, they only amplify the challenge in terms of the amount of variability introduced at the level of configuration, introducing new challenges for application developers (Singh, 2011).

1.2.4 Tradeoffs

With this growing number of linguistic support mechanisms to choose from, we must understand how the approaches compare in terms of the problems they best solve, the level of abstraction they provide and the amount of control they offer to the programmer. The language community does not appear to be converging on one set of primitives for expressing concurrency—quite the opposite. Figure 1.2 demonstrates the magnitude of this with a list of parallel programming environments of the 1990s taken from a presentation at Berkeley Parlab's bootcamp (Keutzer, 2010). Given the current spectrum of languages, we now have a set of overlapping language mechanisms for expressing similar concepts but emphasizing different elements of this problem domain. These variations in representation become problematic to developers attempting to comprehend code bases developed in different languages.

Studies that provide a comparison of existing mechanisms are useful in understanding how to select and apply the best language construct for a given problem. Though many parallel languages are still under investigation, it is important to note the need to consider both quantitative and qualitative assessments respectively, and the nature of the tradeoffs between them. That is, we want to consider both the performance and–although difficult to quantify–the effort required to develop an application for a given approach.

Understanding which linguistic support mechanism best fits the application being developed and the characteristics of the data to be used is not a simple task. The argument as to what kind of library mechanisms best support concurrency goes back as far as 1979 (Lauer and Needham, 1979), with the war of threads versus events. Threads provide preemptive scheduling abilities to interleave work whereas events provide a single, non-preemptive execution stream using event handlers. The dispute continued, with the invited talk by Ousterhout (Ousterhout, 1996) highlighting the propensity for deadlock and corrupt data through synchronization errors. While Ousterhout conceded that threads are more powerful than events, he warned against the inherent complexity. A rebuttal came in 2003 from von Behren (von Behren *et al.*, 2003), arguing that events do not scale and that the complexity of threads can be lessened with compiler support, as demonstrated in Cappriccio's threading model for Internet services (Behren *et al.*, 2003). While each side of the argument has merit, it really only applies in a situation of false concurrency: that is, on one CPU. Even Ousterhout conceded that threads were the only way to achieve true concurrency

AM	DC++	ISIS	Modula-P	pC	distributed Smalltalk
AMDC	DCE++	JAVAR	Modula-2	pC++	SMI
AppLoS	DDD	JADE	Multipol	PCN	SONiC
Amoeba	DICE	Java RMI	MPI	PCP	Split-C
ARTS	SIPC	JavaPG	MPC++	PH	SR
Athapescan-Ob	DOLIB	JavaSpace	Munin	PEACE	Sthreads
Aurora	DOME	JIDL	Nano-Threads	PCU	Strand
Automap	DOSMOS	Joyce	NESL	PET	SUIF
bb_threads	DRL	Khoros	NetClasses++	PETSc	Synergy
Blaze	DSM-Threads	Karma	Nexus	PENNY	Telegrphos
BSP	Ease	KOAN/Fortran-S	Nimrod	Phosphorus	SuperPascal
BlockComm	ECO	LAM	NOW	POET	TCGMSG
C#	Eiffel	Lilac	ObjectiveLinda	Polaris	Threads.h++
“C#in C	Eilean	Linda	Occam	POOMA	TreadMarks
C##	Emerald	JADA	Omega	POOL-T	TRAPPER
CarlOS	EPL	WWWinda	OpenMP	PRESTO	uC++
Cashmere	Excalibur	ISETL-Linda	Orca	P-RIO	UNITY
C4	Express	ParLin	OOF90	Prospero	UC
CC++	Falcon	Elean	P++	Proteus	V
Chu	Filaments	P4-Linda	P3L	QPC++	ViC#
Charlotte	FM	Clenda	P4-Linda	PVM	Visifold V-NUS
Charm	FLASH	POSYBL	pable	PSI	VSE
Charm++	The FORCE	Objective-Linda	PADE	PSDM	Win32 threads
CID	Fork	PiPS	PADRE	Quake	WinPar
Cilk	Fortran-M	Locust	Panda	Quark	WWWinda
CM-Fortram	FX	Lparx	Papers	Quick Threads	XENOOPS
Converse	GA	Lucid	AFAPI	Sage++	XPC
Code	Gamma	Maisie	Para++	SCANDAL	Zounds
COOL	Glenda	Manifold	Paradigm	SAM	ZPL

Figure 1.2: Parallel programming support mechanisms provided by Berkeley Parlab’s annual bootcamp (Keutzer, 2010).

across multiple processing units.

A comparison of MPI and OpenMP surveyed various implementations, including both combined and independent approaches (George *et al.*, 1999). In this study, results showed the MPI implementation ran faster than the implementation with both OpenMP and MPI due to secondary cache misses slowing down execution (George *et al.*, 1999). Additional studies support the result that it may in fact be difficult

to achieve consistent performance gains using OpenMP, due to low-level issues that are not immediately obvious (Krawezik and Cappello, 2006). While highly abstract support mechanisms tend to provide performance gains to some extent, the truth remains that in order to achieve full efficiency, a developer must be exposed to the complexity of the underlying hardware and program effectively to it.

A further performance study comparing MPI, OpenMP and PThreads also confirmed OpenMP requires the least amount of programming overhead, but at the same time was harder to resolve issues, requiring more control in terms of thread affinity and memory locality (Stamatakis and Ott, 2008). Though no one implementation in this study outperformed the others across all tests, it was noteworthy that PThreads and MPI were more portable and flexible, and hence had advantages of sustainability that should be considered an asset. However, it was additionally noted that tight integration with low-level architecture specific issues, such as the number of threads available per CPU as dictated by the hardware design, by far yielded the best performance results.

Contrary results can be found as well, for example in the study of OpenMP compared to PThreads (Kuhn *et al.*, 2000) showed that OpenMP was not only faster from a performance perspective, but the resulting code was of a higher quality. That is, thread-safe data structures were easily implemented, as thread synchronization issues could be automatically handled.

A preliminary study to include OpenCL in a comparison with OpenMP and PThreads revealed that the speedups achieved were roughly proportional to the amount of work involved (Kiemele, 2009). Additionally, the level of implementation detail OpenCL shields the developer from in terms of managing bandwidth bottlenecks appears to be significant. OpenMP was the slowest, but the easiest to implement and still showed competitive performance improvements over a sequential implementation.

1.3 Practices, Patterns and Practicality

In order to refine the scope of this investigation of support strategies for developers in the parallel domain, we narrow the solution space to consider structured software development practices, design patterns and the use of real-world abstractions. A survey of the evolution of software development practices demonstrates the move towards a more iterative and artefact-driven development model in which tools play a

key role (Section 1.3.1). The wide acceptance of design patterns in the object-oriented domain motivated our consideration of this as a solution space for the parallel domain. A deeper investigation of the history of design patterns and an overview of the other programming domains that use them is overviewed (Section 1.3.2). Existing human mental models have been leveraged to convey complex concepts for both program comprehension as well as navigation of a knowledge base. A survey of this use of abstraction as applied to computer related concepts is provided (Section 1.3.3).

1.3.1 Software Development Practices

In *No Silver Bullet: Essence and Accidents of Software Engineering* (Brooks, 1987), a follow up to his advocacy for software development practices (Brooks, 1975), Brooks overviews the contributions of the many research areas working towards the development of quality software, including high-level languages, new paradigms and tools, Brooks is quick to point out that there is no one solution to the problem:

> "There is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity."

Brooks advocated for attention to design supported by prototyping and iterative development, believing that it is the conceptual part of development that introduces the time bottleneck in the development process. Interestingly, Brooks held no hope for hardware to answer the problem, but little did he know that it would amplify that problem:

> "More powerful workstations we surely welcome. Magical enhancements from them we cannot expect."

Further, in Brooks' anniversary edition of *The Mythical Man Month* (Brooks, 1995) he identifies mistakes in the original edition. Specifically, he proclaims Parnas' ideas about of information hiding to provide a layer of abstraction to the developer (Parnas, 1972) to be on the *right* track. We are witnessing this question of the pros and cons of information hiding playing out in the parallel language debate with some languages hiding the complex hardware details for the purpose of simplicity, while others expose these details to allow for full performance gains.

Software development processes have introduced a structure to the creation of software in the form of stages or phases in which intermediate artefacts are produced, where an artefact is a tangible result that is input to subsequent stages. Most current processes are iterative but even these have evolved over time. The much discredited Waterfall model (Royce, 1970) (Figure 1.3), proposes a sequential organization with some variation of the following tasks: *requirements*, *design*, *implementation*, *testing* and *deployment*. Each stage in this model was to be visited once and with the resulting artefact feeding directly into the next stage. Even Royce, who is considered by some to be the creator of the Waterfall Model, pointed out the flaws in this model and that testing would often force a redesign and other similar feedback situations (Royce, 1970).

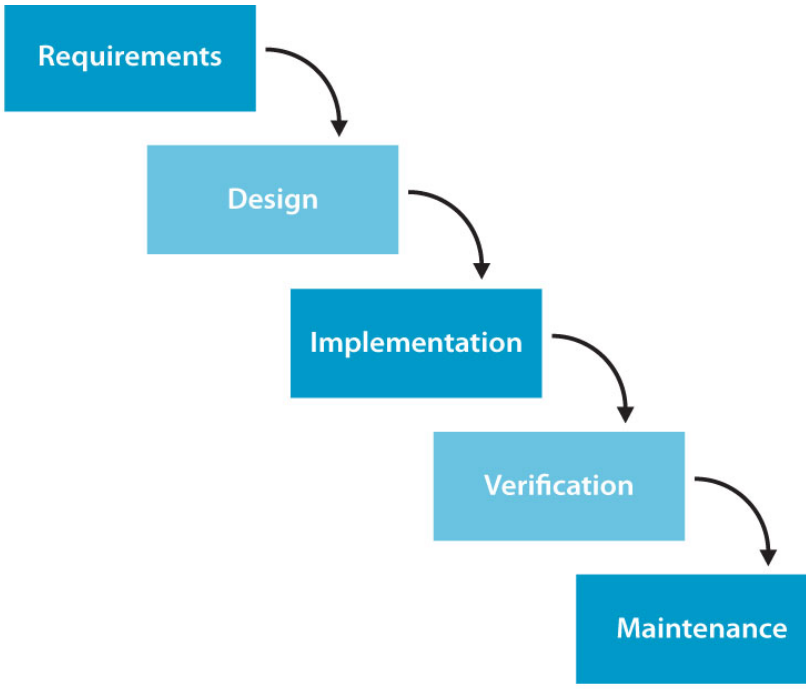

Figure 1.3: The Waterfall software development model (Royce, 1970).

Many alternatives to the Waterfall model were developed to include a more feedback-based approach. One such approach was the Spiral model (Boehm, 1988), an iterative process proposed by Boehm that focuses on a repeated risk analysis phase. One iteration consists of four phases as demonstrated in Figure 1.4 and lasts from 6 to 24 months. In the Waterfall model, each stage is carried out only once with the result

feeding into the next stage, the spiral approach has each stage visited multiple times with the whole previous iteration providing input to the next stage.

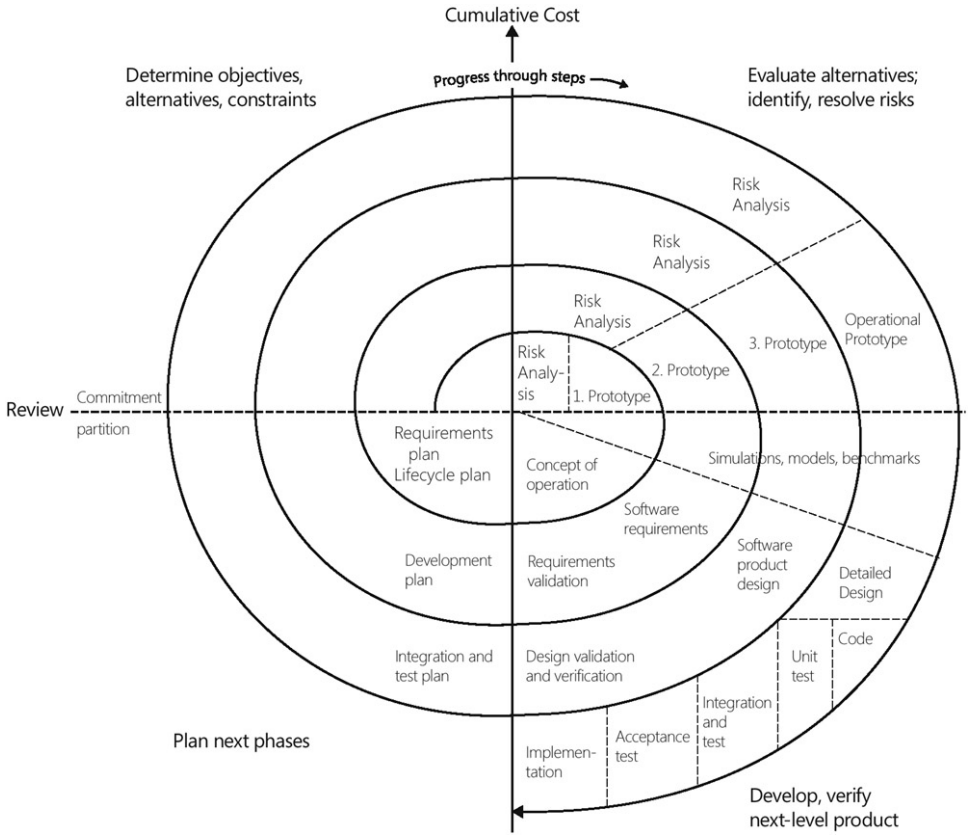

Figure 1.4: Boehm's Spiral software development model (Boehm, 1988).

Agile development (Beck *et al.*, 2001) proposes a similarly iterative process but with the iterations being strictly controlled by feedback as opposed to planning based on risk analysis. The development cycle is in short iterations, tied to face-to-face customer feedback to determine if functional concerns are being met.

The Unified Process (UP) (Jacobson, 1999) is a Software Development Lifecycle (SDLC) model that proposes an iterative and incremental approach. The UP approach is described as *use case driven*, *architecture centric* and *risk focused*. *Use cases* provide a description of application behaviour from an external perspective. That is, how the program will be used and the expected result of those uses. The architectural design of an application serves as a guiding artefact in the development and evaluation of the software, demonstrating the *architecture centric* nature of this model. The *risk focus* forces the development process to deal with finances, safety

and ethics early on in the development process. It is interesting to note how these non-functional requirements start to fit within SDLC models in general. This model is considered effective and helpful in terms of general software development but will it continue to apply for the parallel application development process?

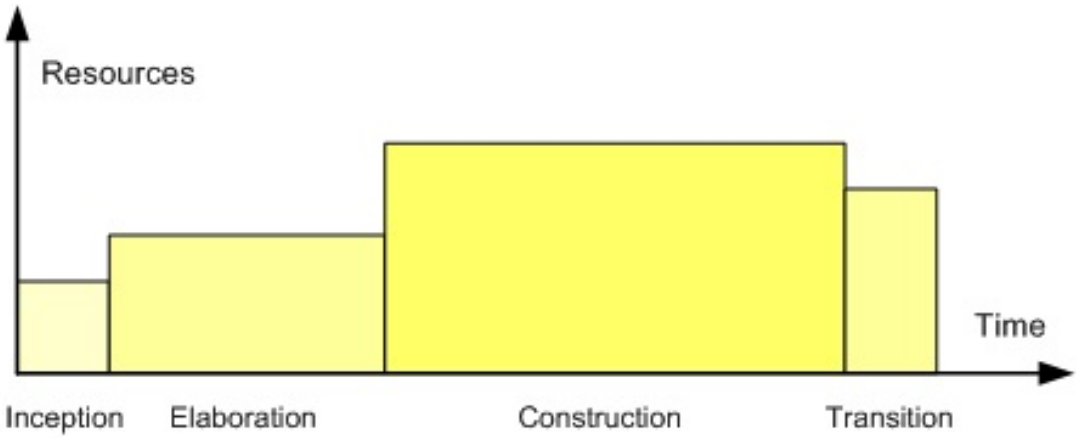

Figure 1.5: Project profile applying UP model (Jacobson, 1999).

Four main phases comprise the UP framework: *inception*, *elaboration*, *construction* and *transition*, which are each divided into time-bounded increments as shown in Figure 1.5. The *inception* phase is short and focuses on project planning including risk identification, core requirements gathering, project costs and schedule. The *elaboration* phase involves more in depth requirements gathering with project planning driving creation of design documents and some initial implementation. The largest phase is *construction* in which the rest of the system is built. Iterations in the *construction* phase are initiated with full use case descriptions and result in an executable release. The final phase of *transition* deals with deployment and iterative refinements based on user feedback.

While from Figure 1.5, the UP model seems sequential like the waterfall model, it in fact includes multiple iterations within each phase. Each iteration in these phases is made up of the different workflow processes: *business modeling*, *requirements*, *design*, *implementation*, *testing* and *deployment*. The UP phases of development are artefact driven, that is, they capture and support changes to static artefacts that in turn are validated or tested through dynamic output. The dynamic result, such as an executable release of the software, provides feedback and forces a reworking of the static artefacts created in earlier workflows. While these workflow processes are

similar to the stages of the Waterfall and Spiral models, they are not required to be carried out sequentially as in the Waterfall model nor within an iteration of the Spiral model. In fact, how much time spent in a specific workflow of an iteration can range from none to a substantial amount depending on the current phase of the process (*inception*, *elaboration*, *construction* or *transition*).

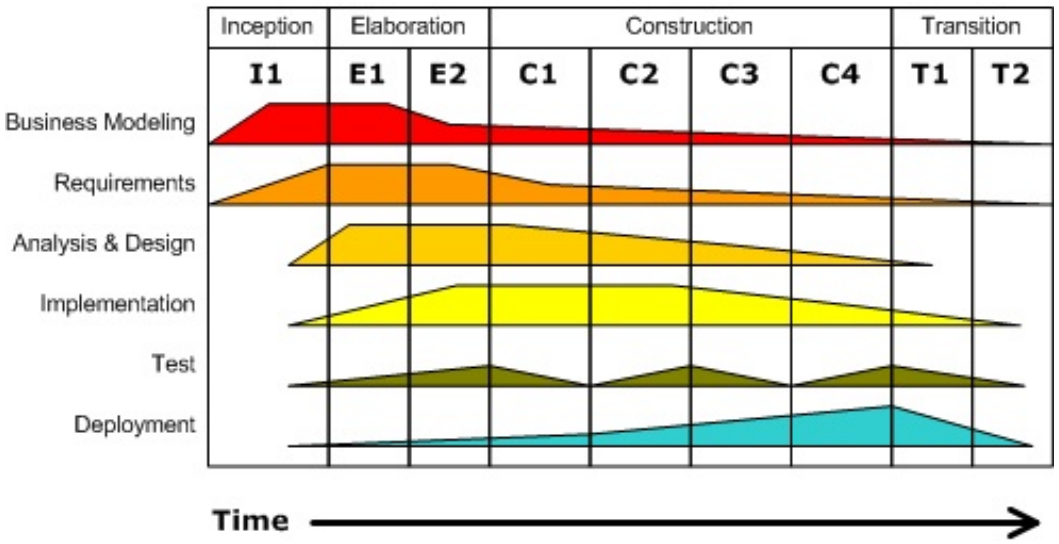

Figure 1.6: Iterative workflows of the RUP model (Jacobson, 1999).

Specific instances of the UP framework, such as the Rational Unified Process (RUP) (IBM, 1988), provide a concrete description of the static structure that makes up the processes. RUP describes each process in terms of *workers*, *activities*, *artefacts* and *workflows*. The workers describe the project participants, their behaviour and responsibilities; the activities describe how things are to be done in terms of creating or updating artefacts. The artefacts are the tangible pieces that are either being used as a reference, modified or generated within the process, with each artefact being owned by a worker. Finally, the workflows describe when the activities will occur with respect to each other, which artefacts will be produced and which workers will be involved. Each iteration in the UP is made up of the different workflow processes: *business modeling*, *requirements*, *design*, *implementation*, *testing* and *deployment*.

Figure 1.6 illustrates how the different workflow processes vary in weight within each of the four phases of the UP and how each phase always involves more than one workflow. Looking at this figure, we can see that not all development iterations in a phase include all workflows. For example, the first iteration in the inception

phase is limited to the business modeling and requirements workflows, whereas the implementation and testing workflows are more prominent in the construction phase.

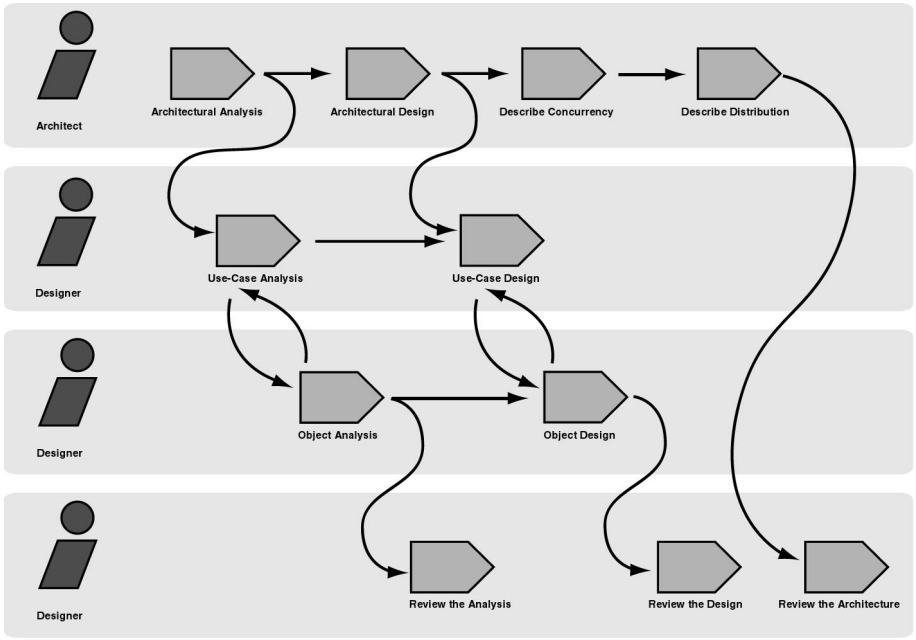

Figure 1.7: Example workflow in Rational Unified Process (RUP) model (Kruchten, 2003).

Figure 1.7 provides an example of a RUP workflow associated with analysis and design. This image provides a visual representation of the key relationships that exist between the worker responsibilities, the artefacts being generated within a workflow and those feeding into subsequent tasks within the workflow. This image demonstrates the true concurrency that exists within the software development process, with multiple people working within a single workflow, developing, using and depending on shared artefacts. This brings us back to the software development challenge of coordinating manpower identified by Brooks. Understanding how to support the coordination of artefact creation and access is imperative to success.

In order to introduce new software development practices and models that take into account both functional and non-functional requirements of parallel applications, such as performance and resource management, we must consider new and existing tools to support these practices through visualisation and dynamic analysis. The following survey of analysis tools and Integrated Development Environments (IDEs) provides context in terms of existing tools to support these SDLC processes (Sec-

tion 1.3.1.1).

1.3.1.1 Tool Support

Static analysis of source code has proven useful to identify and validate correct behaviour that is often difficult to manually assess, such as the nuances of simple locking strategies and race conditions (Engler and Ashcraft, 2003). Due to the amount of code that must be considered and the combinations of events that can cause bugs, several strategies to automate bug detection have been developed. *MUVI* demonstrates promising results for the detection of multi-variable concurrency bugs applying data mining techniques combined with code analysis (Lu *et al.*, 2007), with the focus being on the analysis and results, not the data mining method.

However, in order to better determine issues related to performance and resource utilization, dynamic approaches are required. Standard debuggers allow developers to interact with an executing program in a single run, given a single set of inputs as an artefact. Exceptional system behaviour, such as the results of alternative thread scheduling outcomes, or external dependencies, such as device interactions, is notoriously hard to capture and analyze.

Aspect-oriented approaches (Ansaloni *et al.*, 2010) that aim to address core principles of software engineering through improved modularity have been leveraged to run profiling code asynchronously and buffer it, which has been shown effective on the DaCapo Benchmark Suite (Blackburn *et al.*, 2006). But, in terms of dynamic analysis of parallel applications, several proprietary tools handle both dynamic profiling and performance monitoring (Intel Corporation, 2008b; Red Gate Software, 2008), specifically targeting parallel applications; these tools provide visualisation of performance—often down to a single line of code—and some even provide optimization suggestions.

There are many different tools and techniques that can assist developers in the analysis of dynamic information about a system. Program slicing (Weiser, 1984) assists in the understanding of relationships between statements that span a control flow path. *Time traveling debuggers* (King *et al.*, 2005; O'Callahan, 2008) allow developers to go back in time to explore statements that have been previously executed and possibly changed values. Execution traces have been captured at multiple layers in the system, including hardware (Xu *et al.*, 2003), software (Bhansali *et al.*, 2006) and at the virtualization layer (Dunlap *et al.*, 2002). The primary attraction

of using virtualization is the ability to collect the trace information without the need for instrumentation, resulting in commercial *hypervisors* which handle the coordination details of virtualization, introducing as little as 5% overhead (Chow *et al.*, 2008; Xu *et al.*, 2007). *Tralfamadore* uses OS level virtualization to record long running system behaviour, mapping system events back to program source (Lefebvre *et al.*, 2009). Query languages have been created to assist developers in the analysis of these large system code traces (Martin *et al.*, 2005; Goldsmith *et al.*, 2005).

Other tools target Java code analysis, searching for possibilities of deadlock through graph creation and analysis (Williams *et al.*, 2005). Tools like *KISS* (Qadeer and Wu, 2004) translate a parallel program to a sequential version and then run a sequential model checker on the resulting code, whereas *ZING* (Andrews *et al.*, 2004) is a model checker specifically designed for concurrent programs. Most of these tools require the user to instrument their code with assertions that specify concurrency constraints. The problem with many of these tools is the overhead of manually sifting through false positive/negative results.

Profiling tools, such as the *OProfile* (OProfile, 2010), automatically instrument an executable in order to record system runtime information. In a threaded environment, the execution will often be modified dramatically and threading will even sometimes be removed. Other dynamic analysis tools, such as *Valgrind* (Nethercote and Seward, 2007), are excellent for identifying resource issues associated with cache inefficiencies or memory leaks, and have further capabilities for detecting errors in multi-threaded programs that use POSIX thread primitives. Modern IDEs are starting to include information about dynamic characteristics such as runtime types, the number of objects created, or the amount of memory allocated to methods (Rothlisberger *et al.*, 2009).

IDEs now support navigation of complex code bases allowing multiple code fragments to be simultaneously viewed and edited in novel and interactive ways (Bragdon *et al.*, 2010a,b). Visualization tools have also been shown to be useful for understanding and managing complex configuration issues (Adams, 2008). The ability to modify programming language conventions and mechanisms at a lightweight level, in the browser only, may facilitate programming language innovation without the need to first revamp all other tool support (Davis and Kiczales, 2010). With the parallel paradigm shift, IDE support is moving into this space to provide debugging support for multi-threaded applications. TotalView (Total View Technologies, 2008) provides this support for OpenMP and MPI parallel programs and Microsoft Visual

Studio 2005 (Microsoft, 2008) provides execution of parallel programs with debugging support in the form of multi-threaded breakpoints. This mix of general and parallel-specific IDE support demonstrates how research, in terms of cognitive support for developers, continues to move forward. Drawing from lessons learned and extending existing IDE support to take into account parallel-specific issues provides a starting point for parallel-specific IDEs.

1.3.2 Design Patterns

Design patterns are a widely accepted approach to describing a general solution in the context of a frequently occurring problem in software (Hillside Group, 2010). The applicability of a given pattern in multiple settings has made design patterns one of the most widely accepted abstractions in the design phase of a SDLC. The blueprint of a single design pattern solution makes up just a portion of a program. That is, the structure of the whole program is not provided by a single pattern. This motivates the need to be able to select and compose patterns in order to build an entire application.

Groups of patterns are often presented as a unified catalogue that are grouped categorically, with each pattern individually identifying relationships to other patterns. For example, the Gang of Four (GOF) patterns are grouped into *Creational*, *Structural* and *Behavioural* patterns with each pattern including a section entitled *Related Patterns*. This organization provides a browseable catalogue, that tries to help the reader understand the relationships between patterns and how they might compose. Creating a unified catalogue of patterns is challenging in any domain and much of the difficultly lies in representing relationships between patterns in a centralized form, which is further compounded by natural growth as new patterns are discovered.

Lessons from design patterns developed for issues of concurrency and parallelism in the past provide some foundation for the development of patterns for multi-core architectures. Patterns like *Active Object* (Lavender and Schmidt, 1995) which decouples method execution from method invocation, supporting the interleaving of shared access to a single data object by multiple threads is commonly used in distributed systems. Other published patterns of concurrency include: *Monitor Object*, *Half-Sync/Half-Async*, *Leader/Followers* (Schmidt *et al.*, 2001), which provide programmers with strategies for implementing communication and coordination issues across processes working together, helping to avoid common pitfalls of concurrency. The *Producer-Consumer* pattern (Tanenbaum, 2007) is a commonly used approach

that supports the coordination of asynchronous production and consumption of resources in a shared data structure; this pattern is closely related to the *Scheduler* pattern (Tanenbaum, 2007) which specifies a scheduling policy on threads waiting to consume resources. In an abstract form, many other similar patterns implicitly describe concurrency issues in the context of thread behaviour. The challenge is to investigate this disparate group of patterns in terms of their similarities and differences and how they apply to this new domain of multi-core architectures.

Pattern languages (Alexander *et al.*, 1977) provide structure that lead a user through a collection of patterns. Though individual pattern languages have been successfully defined within smaller subdomains (Chen *et al.*, 2009; Deugo *et al.*, 2001), understanding the relationships between a larger, more disparate set of patterns is more challenging.

The Berkeley Parallel Computing Lab (Berkeley Parallel Computing Lab, 2010) provides an overview of recent efforts within the parallel community to provide such a pattern language (Keutzer and Mattson, 2010; Johnson *et al.*, 2009, 2010). This pattern language, initially called *Our Pattern Language (OPL)*, began with a simple four-layered approach illustrated in Figure 1.8 in which many of the individual design patterns are not complete. The patterns that have been developed have adopted a standardized format comprised of sections including: *problem*, *context*, *forces*, *solution*, *related patterns*; each pattern is assigned to one of the five categories: *structural*, *computational*, *algorithm*, *implementation*, and *concurrent execution*.

While this structure is beneficial in terms of grouping patterns by purpose and generality, it does not yet fully provide clarity in terms of the relationships between patterns. Current efforts within the community are now focusing on *methodological* patterns (Johnson, 2009; Mattson *et al.*, 2004) to further guide users through this framework, capturing the fine-grained relationships both within and between these proposed layers.

The newest version of Berkeley's *Pattern Language for Parallel Programming (PLPP)* is trying to address this issue with a much more fine-grained, control-flow type of structure as illustrated in Figure 1.9 (Berkeley Parallel Computing Lab, 2010). The intent is to guide developers through pattern selection at the various levels of design. Patterns are grouped by design decisions like *choosing a high level structure*, *identifying key computation patterns* and *choosing a concurrent approach*. In addition, this newer version of the pattern language acknowledges the lower-level issues of efficiency that must be dealt with by the programmer. Although this approach

(notes) (Glossary)(Blogs) (Pattern Template) (Pattern Abstract) (Pattern Workshop) (Pattern in Education)

Applications			
Structural Patterns		**Computational Patterns**	
Agent and Repository	Pipe-and-filter	Backtrack Branch and Bound (notes)	Monte Carlo Methods(notes)
Arbitrary Static Task Graph	Process Control	Circuits(notes)	N-Body Methods
Iterative_refinement		Dense Linear Algebra(notes)	Sparse Linear Algebra
Event-based, implicit invocation		Dynamic Programming doc(notes)	Spectral Methods
Layered systems		finitestatemachine.pdf	Structured Grids (notes)
Map reduce		Graph Algorithms	Unstructured Grids
Model-view controller		Graphical Models	Sorting
Parallel Algorithm Strategy Patterns			
Task Parallelism (notes)	Discrete Event	Geometric Decomposition(notes)	Non-work-efficient Parallelism
Recursive splitting(notes)	Pipeline doc(notes)	Data Parallelism	Speculation
Implementation Strategy Patterns			
SPMD	MasterWorker	sharedqueue.pdf	Distributed Array
Strict-data-par	LoopParallelism	Shared Hash Table	shared data
ForkJoin	BSP		memory parallelism(notes)
Actors	Task Queue		
Graph Partitioning(notes)			
(Program Structure)		(Data Structure)	
Concurrent Execution Patterns			
MIMD	Thread Pool	Message Passing	P2P Sync
Task Graph	Speculation	Collective Communication	Collective Synchronization
SIMD	Data Flow	Mutual Exclusion	Transactional Memory
DigitalCircuits (notes)			
(Advancing Program Counters)		(Coordination)	

Figure 1.8: Berkeley Parlab's initial pattern language structure: *Our Pattern Language (OPL)* (Keutzer and Mattson, 2010).

narrows the scope for pattern selection, it does not yet provide a way to compare across patterns for both selection and composition.

Compositional opportunities between patterns must be made available through combinations of existing patterns that can share a common objective. As Alexander originally described (Alexander *et al.*, 1977; Alexander, 1979), a pattern language should provide more than an infrastructure for browsing a set of patterns:

> "Thus, as in the case of natural languages, the pattern language is generative. It not only tells us the rules of arrangement, but shows us how to construct arrangements—as many as we want—which satisfy the rules." (Alexander *et al.*, 1977)

While Alexander demonstrates the benefits of awareness of pattern compositions in a building architecture domain, a clear understanding of how to select and compose software patterns is still a challenge in both the general and parallel domains.

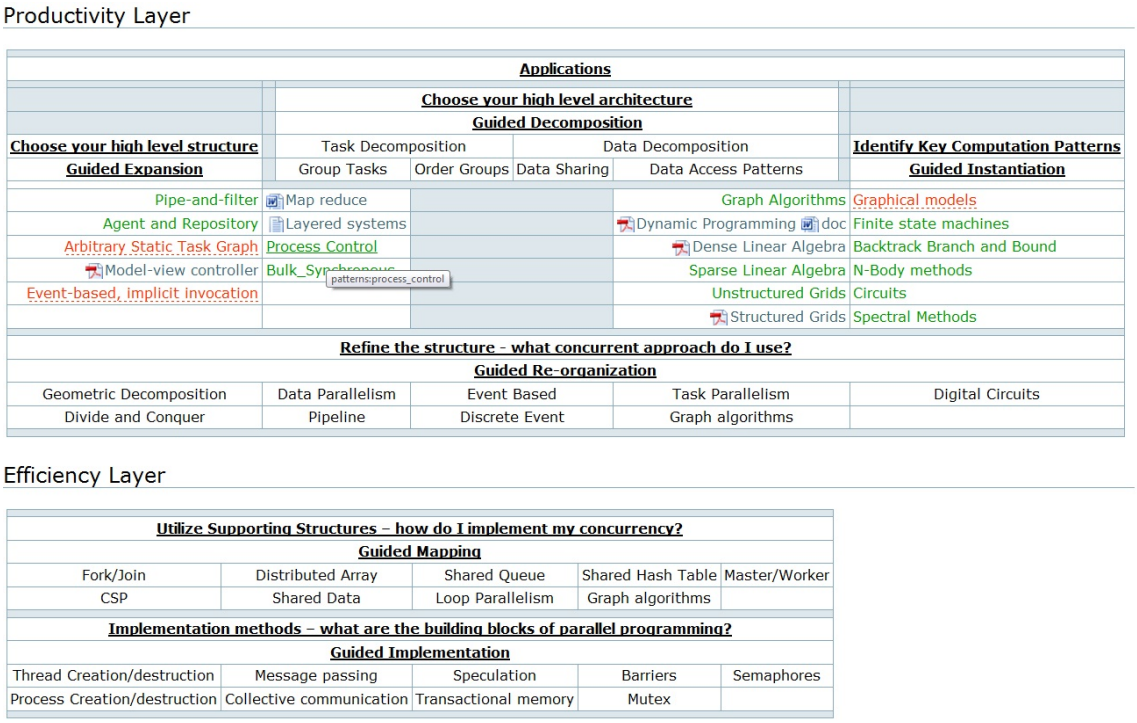

Figure 1.9: Berkeley Parlab's current pattern language structure: *A Pattern Language for Parallel Programming* (Berkeley Parallel Computing Lab, 2010).

Growth in terms of the number of new patterns being developed makes management and navigation of groups of patterns challenging even within the coarse-grained categorization provided by pattern languages. Alternate classifications to reduce the number of fundamental design patterns to consider have been combined with more systematic and concrete class libraries or families of patterns to make patterns both more accessible and traceable to code (Agerbo and Cornils, 1998). Other strategies, such as *Design Pattern Rationale Graphs* (Baniassad *et al.*, 2003), reconcile design with source to aide developers to make changes to source that are in keeping with an existing design. A graphical representation of both source and design patterns are linked by edges through an intermediate level representing relationships between the source and patterns. Navigation is bidirectional between source and design by way of queries.

It appears that the space of embarrassingly parallelizeable problems is also positioned as low hanging fruit for industry solutions, from companies like Microsoft Research (MSR) and Google. The *parallelizeable loop* described by MSR (Leijen and

Hall, 2007), in which data and computation can be split at iterations of a loop such that no dependencies exist between them are now being described as a pattern. Loops in which each iteration modifies or accesses the same data is not as easily parallelizeable due to race conditions. Though locking solves this problem, performance can suffer. Similarly, MSR's *parallel aggregation* pattern in which each thread maintains a local copy of the shared data and only adds this local total to the global data at the end of the loop. This pattern provides a more general solution and encompasses the MapReduce pattern within it. This lack of consensus for pattern naming and development across communities makes navigation of this solution space challenging for a developer trying to become experienced in this domain.

Google's implementation of the MapReduce pattern (Dean and Ghemawat, 2004) focuses largely on fault tolerance leveraging the Google file system and again takes advantage of, essentially, an embarrassingly parallelizeable approach to distribute computation across multiple computers or nodes. The *map* phase partitions the computation and data to the worker nodes passing the result back to the *reduce* phase which aggregates the result into a single output. Newer frameworks, such as Apache's Hadoop (White, 2007), an implementation of the MapReduce pattern described above, are designed to support parallel computations over massive data sets and have achieved remarkable reliability by replicating the data across multiple hosts like the Google implementation, but offer potential mainstream appeal. These frameworks, while shielding developers from underlying complexity handled by the system, limit the control of fine-grained synchronization details by restricting the programming model.

The linkage between original GOF patterns and issues of concurrency is beginning to be addressed with the Panini Project (Rajan *et al.*, 2010). This project attempts to integrate encapsulated portions of concurrency through a framework while adhering to the modular nature of components and interactions set up in the GOF patterns. A case study applies this framework to all 23 GOF patterns, 18 of these patterns maintained their modularity with the addition of concurrency.

1.3.3 Real-World Scenarios

Kristen Nygaard, co-creator of Simula (Dahl and Nygaard, 2003, 1966) and ultimately object-oriented (OO) programming, developed this paradigm top down, taking inspiration from real-world surroundings. Nygaard leveraged this affinity to real-world

scenarios to teach complex programming concepts. Nygaard conveyed object-oriented concepts at an abstract level, tying to them to existing mental models such as the restaurant scenario. He says in this presentation:

> "Teaching object-oriented programming thus must begin with exposing students to complex situations that only may be mastered by understanding and using these concepts." (Nygaard, 2001)

This quote from Nygaard is in the context of a presentation on the pedagogical approach to OO programming leveraging a restaurant scenario, shown in Figure 1.10. This rich representation of a real-world scenario demonstrated for Nygaard the complexity of how, although each table and each individual was a similar entity, they each had their own set of unique circumstances. This image not only helped in the explanation of object-oriented issues, it introduced concepts of concurrency that aligned with Simula, where individual scenarios were playing out individually within the big picture.

Likewise, we use existing mental models of parallelism in our day-to-day lives without giving much thought to what we are doing. We believe these conceptual models can be leveraged to develop explicit reasoning strategies for complex issues associated with parallel development. For example, a group of people assigned household chores can can distribute and manage the workloads. These everyday occurrences of parallelism range from highly parallelizeable, requiring seemingly little coordination between participants, to those requiring more explicitly complex coordination in order to achieve the most even sharing of workloads.

Folding and putting away laundry is a highly parallelizeable job, in which the large pile of laundry can be split up between individuals in a group: each person works on folding in isolation on their own sub-pile. There is no communication necessary between the workers once the workload has been divided; an embarrassingly parallelizeable problem.

But even more complicated scenarios are handled in our every day lives. Take the example of two people asked to clean a set of dirty dishes which is described as three subtasks: 1) washing, 2) drying and 3) putting away the dishes. The challenges so real to concurrency such as load balancing, resource sharing and coordination of work across heterogenous elements do not seem to cause the same level of difficulty in real-world scenarios.

Figure 1.10: Object-oriented constructs represented in Nygaard's restaurant scene (Nygaard, 2001).

The difference between these two ends of the spectrum appears to lie in the issue of communication that facilitates the organization of the job across workers. In real life, verbal communication facilitates the protocol resolution and agreement, while lightweight, visual and almost subconscious communication establishes progress and supports fine-grained coordination of tasks. Communication within a computer application is not as freely achieved; it must be explicitly determined and align with newly defined protocols. Scenarios such as these have been leveraged for pedagogical purposes in efforts to introduce complex computer science concepts such as recursion and concurrency to grade school students (Gunion, 2009).

One possible approach to assisting developers to develop a conceptual model of a problem domain is the use of higher level abstraction such as that of an *ontology* to correlate complex concepts using approaches based on ontology mapping. Ontolo-

gies were first leveraged in computer science by the artificial intelligence community when McCarthy used this technique to model knowledge systems (McCarthy, 1980), later for supplying inter-operability standards (Neches *et al.*, 1991), and finally in terms of specifications for conceptual models (Gruber, 1993). The practical uses of ontologies to represent concepts and entities with properties and relationships within the domain of Computer Science has been well studied, particularly for knowledge transfer (Gruber, 1995). Repositories such as the Protegé Ontology Library (Protegé, 2010) contain ontologies for everything from cell processes to tourism.

Generally, an ontology models a domain as a set of concepts and relationships (Gruber, 2009). Representation of concepts in an ontology typically relies on definitions that allow us to categorize domain knowledge and establish membership within categories (Murphy, 2002). Though this approach has fallen out of favour within cognitive psychology practitioners due to the inability for typical definitions to capture enough information in a way that easily determines membership (Hampton, 1979), it is the basis for creating logical expressions that can be processed in an automated fashion.

At a more detailed level, constraints and axioms can formally shape the meaning of concepts and relationships (Tun, 2006), providing better support for practitioners attempting to comprehend information within a domain. Languages for developing ontologies include Protege (Protegé, 2010), Knowledge Interchange Format (KIF) (Genesereth *et al.*, 1992), and Web Ontology Language (OWL) (OWL, 2010). In particular, OWL has been recommended as a standard for developing ontologies for the Semantic Web—an effort to integrate data to create a semantically linked database for the Internet (Palmer, 2001). Creating standards for data formats, for example OWL versus RDF (WC3 – Semantic Web, 2004), enforces consistency by supporting data inter-operability. Within an ontology, a *schema* is a representation of entities that defines a set of possible instances, and mappings can exist to correlate entities in different schemas (Bernstein and Melnik, 2007; Shvaiko and Euzenat, 2008). Entities in different schemas have complex relationships beyond equivalence, and include entities subsuming one another or even overlapping.

But how do we know a *good* ontology from a *bad* ontology? There are some diagnostic tools, such as Chimaera (McGuiness *et al.*, 2000) that check for logical correctness, identifying common errors in common practices. The critical point that we need to stress here is that the quality of an ontology can only be established by the ability to use it with applications it was designed to support (Noy and McGuiness, 2001). Though this process could most likely be semi-automated at best, requiring

some manual intervention, recent results hold promise that this approach is feasible from a user's perspective (Falconer, 2009).

1.4 Dissertation Organization

The thesis of this research is that software development practices and integrated environments can better support parallel development with a model that structures the acquisition and tracking of tangible artefacts key to the parallel development cycle. Systematic processes and parallel-specific conceptual models can provide support for analysis and comparison within and across these artefact from an implementation level to a level of abstraction that supports reasoning about these artefacts as a knowledge base.

In order to support this thesis we investigate how software development practices and developer resources need to be redefined to target parallel-specific issues and how a parallel-specific conceptual model can tie these practices and resources together. Parallelism or true concurrency imply execution occurring on separate processing units simultaneously. We investigate the use of real-world scenarios in the form of pedagogical activities and develop an ontology that would map to various representations of a software solution as a knowledge base to support both comprehension and navigation between representations. Specifically, we identify the following research objectives to structure this work, with each being addressed in the subsequent chapters of this dissertation:

- **O1.** Determine what is easy/hard about parallel software development.
- **O2.** Discover artefacts key to the parallel software development process.
- **O3.** Discover practical approaches for analysis and comparison of key artefacts.
- **O4.** Discover a conceptual model representative of solutions to parallel problems.
- **O5.** Use the conceptual model developed to describe and map to multiple artefacts in a uniform way.
- **O6.** Propose a framework that supports the analysis and comparison of key parallel software development artefacts identified.

- **O7.** Evaluate the framework and development process in the context of parallel software artefact examples at the conceptual and implementation level.
- **O8.** Evaluate the framework and development process in the context of existing optimization strategies.

This work is organized into four parts: an introduction to the problem domain (Chapter 1), a proposed model to support parallel development (Chapter 2), identification of ways to track and analyse parallel software artefacts (Chapters 3 through 5) and application and evaluation of the proposed development model (Chapter 6). An organization in terms of the contributions of this thesis is outlined in Figure 1.11.

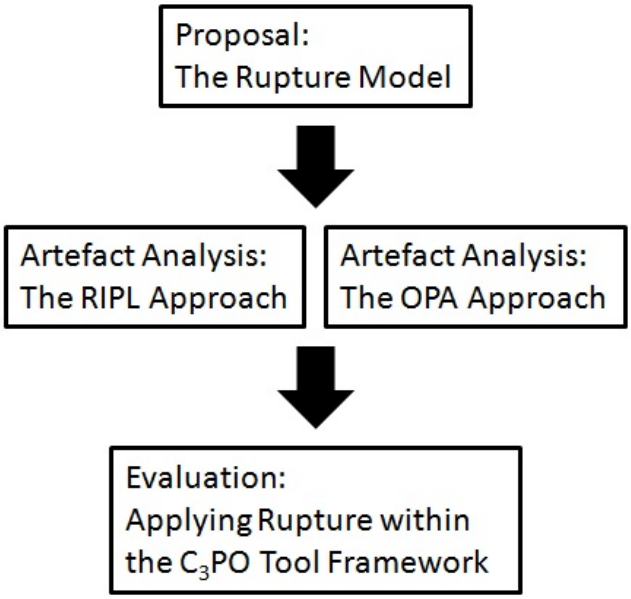

Figure 1.11: A visual representation of the outline of this research that makes up this dissertation.

Chapter 1 introduces relevant background to the parallel programming domain and related work in the area of support strategies for parallel software development. Chapter 2 leverages existing code bases to identify unique challenges and key artefacts within the parallel development domain, and proposes an artefact-driven, parallel-specific development model. Chapter 3 begins the investigation of ways to track and analyse software artefacts with design patterns, followed by Chapters 4 and 5 which develop and apply a parallel ontology as a layer of abstraction on top of software artefacts. Finally, Chapter 6 evaluates the proposed parallel development model in the

context of IDEs, applying the methodologies and abstractions introduced in Chapters 3 through 5. A summary of the contributions of this research and a discussion of future work concludes this thesis in Chapter 7.

Chapter 2

The Rupture Model: Discovering Artefacts and Relationships Critical to Parallel Software Development

Software development processes provide structured guidance to software developers in terms of activities and, ultimately, the creation of a product. This chapter lays the foundation for subsequent chapters by investigating existing processes and identifying areas of significance that warrant further exploration in terms of support for parallel application development. This research has been vetted and published in the software engineering community (Gibbs *et al.*, 2008).

Targeting research objectives **O1** and **O2** of the thesis, we leverage existing code bases in an analysis to understand challenges in parallel software development and discover artefacts key to the parallel software development process. In support of these objectives, this chapter specifically addresses the research question:

> *What does a software development model for the parallel domain need to take into account that existing models do not?*

2.1 Introduction

As outlined in the related work section (Section 1.2.3) the application programmer is about to be swamped with a deluge of options for programming languages, libraries,

frameworks and patterns all designed to assist the inevitable need to parallelize new and existing code bases. Transforming applications from sequential to parallelized versions requires changes that are rarely well contained. Instead, they causally cascade through multiple artefacts—a change in platform (hardware or OS) can force changes to source, causing the corresponding configuration or build to change. Subsequent changes in dynamic characteristics must be tracked, which may call for a change in profiling infrastructure, which may consequently impact memory footprint, adversely affecting performance, and so on. Regardless of the approach or level of abstraction provided, these programmers require processes and structured workflows to assist the development of parallel code.

Software Development Life Cycle (SDLC) models (Kruchten, 2003) provide developers with structured processes for the design, development and testing of software, with each phase generating a set of artefacts or tangible deliverables. Most current models support an iterative and incremental approach, such as the Unified Process Model (Jacobson, 1999), and have structured modern software development practice. Though parallelism manifests itself as a seemingly innocuous, non-functional requirement associated with performance, the ramifications of evolving an existing code base often requires changes to design, implementation, testing and maintenance workflows. Existing workflows do not explicitly take into account the inherent complexities of the dynamic consequences associated with parallel development. The fine-grained and wide-spread modifications and optimizations inherent in parallelization ultimately are challenging to structure within current practices (Brownsword, 2011).

This work investigates systematic workflows structured around key causal relationships stemming from the dynamic impact of changes necessary for parallelization. Much like existing workflows of the SDLC models, we are proposing an artefact driven feedback model, where artefacts in existing SDLC models are some form of deliverable that feeds into the next phase or iteration. In this case, the artefacts are sets of both static and dynamic changes and their relationships to each other that impact decisions for further change. This feedback model, which we call *Rupture*, links change artefacts in *design*, *platform*, *source*, *configuration*, *profiling* and *workload* to their corresponding impact on the *dynamic* characteristics of the system.

We identify the necessary artefacts and the causal relationships between them from existing code bases including the NAS Parallel Benchmark (NPB) suite (NASA Advanced Supercomputing Division, 1999). The motivation to track changes to these artefacts within a SDLC model is investigated in the context of three code

bases: Harmony Portability Library (Apache, 2007a), NPBs and the OmpSCR benchmarks (Dorta *et al.*, 2005). We believe a structured workflow, informed through artefact tracking, could provide application developers with information necessary to make informed development decisions.

The remainder of this chapter identifies artefacts relevant to the parallel development process (Section 2.2), followed by an evaluation of the Rupture model in three different parallel code bases (Section 2.3), and concludes with a summary of the contributions made (Section 2.4).

2.2 Proposal

Over the years, software development models have moved towards an iterative approach, yet they are still focused on static artefacts; success or failure is determined by functional requirements dictated by input, dynamic behavior and final output. Programming for multi-core environments requires additional consideration of non-functional requirements through which dynamic resource utilization on a given architecture often determines success or failure.

Dynamic characteristics are key in a multi-core domain and result from very distinct yet subtle relationships between software and hardware in parallel systems. In particular, it is now critical to be able to reason about hardware utilization and contention in light of different emerging paradigms for software parallelization. Feedback loops of causal relationships are intricate and increasingly important for mainstream programming. Perhaps specific patterns for best practices for resource utilization will emerge over time, as programmers get more experience with the precise tradeoffs involved.

The fine-grained and wide-spread modifications and optimizations inherent in parallelization ultimately are challenging to structure within current practices.

2.2.1 Identifying Artefacts and Relationships

In order to get a clearer understanding of the specific artefacts involved, take for example the NAS Parallel Benchmarks (NPB) (NASA Advanced Supercomputing Division, 1999), which consists of eight programs designed to evaluate performance on parallel architectures. Collectively, they represent essential parallel designs that cover a spectrum of computation and data movement characteristics that typically

create resource contention and utilization challenges in parallel systems.

The results of running the original benchmarks on a wide variety of platforms over several years were reported for over 25 different platforms (Bailey *et al.*, 1994). Within each of these 200 sets of test results, the number of processors was varied, and a comparison was provided for *Class A* versus *Class B*, which differed in the size of their principal arrays. A sample of the benchmarks as run on a single processor is shown in Figure 2.1.

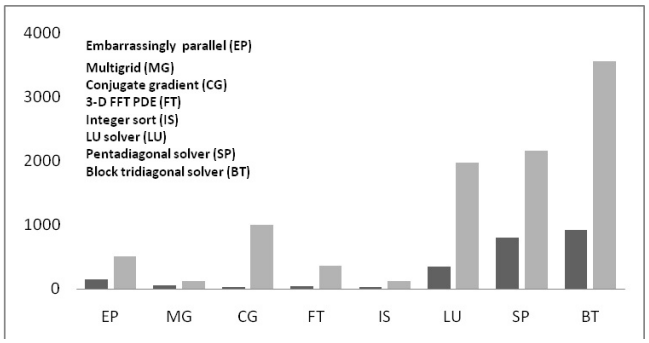

Figure 2.1: Comparison of Class A and Class B performance results measured in seconds on a single processor of a Cray Y-MP (Bailey *et al.*, 1994).

The developers of the benchmarks periodically release new results and further solicit additional results from the community at large. Each submission is required to include the following items in order to completely define a parallel application:

1. A detailed description of the **hardware and software configuration** used for the benchmark runs.

2. A description of the **implementation and algorithmic techniques** used.

3. **Source listings** of the benchmark code.

4. **Output listings** from the benchmarks.

As these benchmarks have withstood the test of time, we derive from this example the following list of artefacts that are (sometimes subtly) causally related to the **dynamic characteristics** of a code base: *design*, *source*, *configuration*, *platform*, *profiling* and *workload*. Table 2.1 provides a summary of these artefacts and a mapping of how they were extrapolated from the NPB requirements.

Table 2.1: Artefacts of interest to parallel developers extracted from NPB submission requirements.

NPBs: Required Items	Example Artefacts
hardware configuration software configuration	platform workload configuration
implementation techniques algorithmic techniques	design
source listings	source
output listings	profiling dynamic

We further propose that management of the software lifecycle requires explicit efforts to structure workflows around the *causal relationships* between these artefacts. In particular, we hope to assist developers in reasoning about changes that crosscut multiple artefact types and cause corresponding dynamic changes, such as system performance and resource utilization.

Table 2.2 provides a summary of the artefact sets, some tangible examples of each and the challenges that are encountered within each set. The last column of this table lists the relationships between artefacts, specifically listing the artefacts that are impacted by the dominant artefact set (Column 1) of the given row. The following discussion provides the reasoning behind the population of this table with a further analysis provided later in the chapter to evaluate this proposal (Section 2.3).

Design artefacts can contain documentation including pattern-specific information, pattern catalogs or results from pattern mining an existing code base. Application design directly impacts the implementation, which plays out within the source and the configuration of an application. Any change to design has the potential to ripple through these artefacts. The analysis of the Harmony Portability Library in Section 2.3.1 demonstrates this characteristic with design decisions and the impact that a change to this design would have on the implementation, forcing subsequent changes to both the *source* and the *configuration* artefacts.

Source, including both source and header files, represent a substantial portion of an application. Much of the complexity of parallel application development is associated with the parallel language mechanisms used within the source. Changing a single line of source can change the workload of an application and the configuration of global

Table 2.2: Parallel artefacts, their challenges and relationships to each other.

Artefact Sets	Example Artefacts	Challenges	Relationships: other artefacts impacted
Design	pattern languages, design patterns	documentation, cataloguing, pattern mining	source configuration
Source	source files, header files	complexity	configuration workload
Profiling	printfs, DEBUG, ompP GNU gprof	serialization, inconsistency, Heisenberg effect	source configuration
Platform	runtimes installation, number of cores, memory model	information not always readily available	design source configuration
Configuration	config files, command line arguments	complexity, redundancy	source profiling workload
Workload	input files, command line arguments	data usage patterns, number of users	design source configuration
Dynamic	performance, memory usage, cpu usage	accurate measurement	impacts all other artefacts in an iterative feedback loop

fields simultaneously. The OmpSCR code base analysis of Section 2.3.3 provides examples of changes to *workload* and *configuration* settings that are made within *source* artefacts.

Profiling can be introduced to an application through the use of tools such as OmpP (Furlinger *et al.*, 2005) and GProf (Fenlason and Stallman, 1998) or even manual introduction leveraging conditional compilation support of `#ifdef` or instrumented `DEBUG` statements within the source files. The challenge with some existing profiling tools is that they sequentialize the execution, negating the impact of the parallelism introduced. In addition, the monitoring of any application must be cognisant of the dynamic overhead or impact that can be incurred by profiling, commonly known as the *Heisenberg effect* (Heisenberg, 1927). In Section 2.3.2 the analysis of the NPBs shows the implementation of profiling within the source code as well as in configuration settings demonstrate the impact of a change in profiling on both *source*

and *configuration* artefacts.

Platform-specific variability is a growing issue with the increasing number of architectures and library support mechanisms introducing different runtime installations, the number of cores or different memory models. These changes in platform are addressed with changes ranging from tailoring configuration options, making changes within source even forcing changes to the underlying design. The analysis of Harmony in Section 2.3.1 demonstrates how adding support for a new underlying operating system would force changes to the *design*, *configuration* and *source* artefacts. As well, the impact of more fine-grained changes in platform and the changes they force to *configuration* are highlighted in Section 2.3.2 in the NPBs.

Configuration settings are arguably one of the most challenging artefacts to track, due to the many ways they can be set in configuration files, flag settings within source and through command line arguments. A change to a configuration setting can in turn mean changing source code and can control *profiling* or change the size of the *workload*. Section 2.3.2 highlights configuration setting changes in the NPBs within a source file demonstrating the impact on *source* artefacts. The control of *profiling* and *workload* within source configuration is also highlighted in this section.

Workload, similarly, can most commonly be set up in configuration files, with command line arguments, within source or be dynamically determined. As mentioned above in the *configuration* discussion, *workload* can be changed within the *source* and a similar result is shown in the OmpSCR code base analysis in Section 2.3.3. The NPB analysis in Section 2.3.2 demonstrates how a change in workload can require a change in configuration of stack size to accommodate different workloads.

The *dynamic* result of an application can be impacted by a change to any one of the above discussed artefacts. Be it performance or resource utilization, this *dynamic* result can warrant subsequent changes to any of the other six artefacts. The subsequent analysis of this proposed model will focus on the static artefacts through an examination of parallel code bases with a focus on the intricacies of these artefacts.

2.2.2 The Rupture Model

Workflows need to incorporate the ability to explore subtle feedback loops and causal relationships between changing artefacts in this intense development process. Figure 2.2 highlights these relationships in terms of changes, or Δs. For example, a $\Delta platform$ would mean a change to hardware or software infrastructure other than

the source of a code base (for example, an OS, or even runtime support). Δ*design* would be a change in the algorithm, accompanied by corresponding changes in the code base. Rossi's Master thesis investigated the optimization of Transmission Line Matrix (TLM) methods to utilize the highly parallel architecture of GPUs (Rossi, 2010). This research demonstrated, through an iterative case study, that these coarse-grained changes can have a monumental impact on the dynamic result of a program.

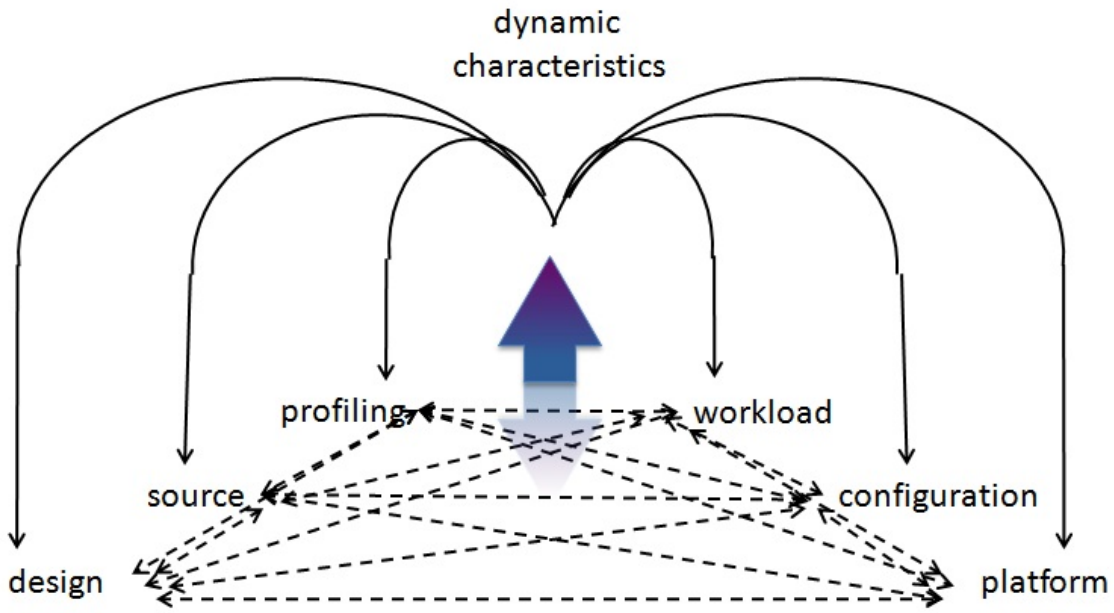

Figure 2.2: Rupture: Artefacts and Relationships

To narrow the problem space, we focus on the following fine-grained artefacts and their causal impact on one or more dynamic characteristics (such as performance, memory usage, resource utilization and contention):

- Δ*configuration*: A change to build or runtime options.
- Δ*source*: A change to source of a code base, including header files.
- Δ*profiling*: A change in profiling infrastructure used specifically for dynamic assessment.
- Δ*workload*: A change in workload/input.

Finally, completing the feedback loop:

- $\Delta dynamic$: A change in a dynamic characteristic that in turn may precipitate many more related changes to $\Delta platform$, $\Delta design$, $\Delta configuration$, $\Delta source$, $\Delta workload$, and $\Delta profiling$.

In terms of integration into the existing UP SDLC model outlined in Section 1.3.1, we believe the Rupture workflow could fit within the construction phase and integrate with the existing analysis and design, implementation and testing workflows. The feedback-based approach of Rupture applies within the iterative nature of the four phases of the UP model and the change artefacts that comprise Rupture align with the existing, artefact-driven UP workflows. Specifically considering the RUP model and the parallel nature of its workflows illustrated in Figure 1.7 of Section 1.3.1, dynamic artefacts, and their correlated design and implementation change artefacts will provide input for inter- and intra-workflow iterations. For example, a decrease in performance identified within a testing workflow can be investigated by feeding the artefacts (ie. tracked changes to configuration and source) back into implementation and design workflows.

The integration of feedback loops between sets of changes, $\Delta i \leftrightarrow \Delta j$, will most likely require system specific tools, but each of these tools can share common strategies. That is, the variability in everything from build systems to profiling strategies can be smoothed into a consistent development process to facilitate a common strategy for viewing and cataloguing changes to sets of artefacts. The extensive amount of information that will accumulate in this type of process from these many artefacts amplifies the need for tools to support both change tracking, analysis and visualisation. We evaluate the proposed artefacts of Rupture in a survey of several code bases, establishing representative examples of both instances of the artefacts involved, and the relationships between sets of changes to them.

2.3 Analysis: Investigating Artefacts and their Relationships

Managing and tracking cascading sets of changes associated with dynamic characteristics is difficult because the artefacts are often intertwined and change sets are the result of tight feedback loops. In order to support the development of best practices and tools to support structured workflows for this domain, we start by trying to gain a better understanding of the artefacts involved and relationships between changes.

Here we consider examples from three code bases, the Harmony Portability Library (Apache, 2007a), NPBs (NASA Advanced Supercomputing Division, 1999), and OmpSRC (Dorta *et al.*, 2005) that each contain varying levels of parallel constructs. Each of these C-based software projects leverage the C-Preprocessor (CPP) (Kernighan and Ritchie, 1978; Ernst *et al.*, 2002) for conditional inclusion of code at compile time. The CPP supports the compile time inclusion of source files using `#include`, constants with flag settings and functionality with macro definitions using `#define` and the conditional compilation of code segments wrapped in `#ifdef`—`#else`—`#endif`.

Within each of the three software projects we identify individual *Rupture* artefacts sample listings followed by examples of the causal relationships between these artefacts with a summary of how these examples support the proposed Rupture model (Section 2.2). This analysis uncovers two types of artefact relationships: 1) one artefact strongly influences another artefact—where a change to one artefact forces an implicit change to another and 2) one artefact is embedded in another artefact—where a change to one artefact introduces an explicit change to the artefact it resides in. While both relationships introduce challenges in terms of tracking changes, the implicit impact a change to one artefacts can have on one or more other artefact is the feedback issue that Rupture looks to address.

2.3.1 The Harmony Portability Library

Apache's Harmony (Apache, 2007a) is a 23 kSLOC (kilo-Source Lines of Code) open source software project that provides a Java runtime with class libraries and tools that support 4 versions of the Windows platform and 11 versions of Linux. Harmony leverages preprocessor directives heavily to achieve a highly configurable Java Virtual Machine (JVM); this is a characteristic common to many C-based implementations and has proven to be useful for conditional code inclusion prior to compilation (Ernst *et al.*, 2002; Hu *et al.*, 2000). Table 2.3 overviews the heavy usage of conditional compilation in the Harmony open source Java SE project. System-wide, there are 261 different flags, affecting more than 400 files. More specifically, 25 flags are present in more than 5 of the 682 files in the system. Roughly 60% of the system, (404/682 files) contain at least one conditionally compiled code block. Understanding the impact of a change to a single flag setting is challenging when it can be affecting more than 5 of the the 404 different source files, especially with many of these flag settings being associated with architecture configurations and optimization choices in

terms of choosing a parallelization strategy.

Table 2.3: Conditional compilation based on CPP flags settings in the Harmony JVM.

Metric	Harmony
Conditional compilation flags	261
Flags affecting more than 5 files	25
Flags affecting more than 10 files	5
Files (.c and .h only)	682
Files containing CPP conditionals	404

As a general design principle, Harmony attempts to modularize platform-specific behaviour (ie. opening a file or a socket) behind a well-defined API boundary, such as that of the Apache Portable Runtime (Apache, 2007b), in the Harmony Portability Library. The use of a function-based API enables a higher degree of selection or replacement of functionality at link-time, load-time and runtime. In terms of load-time options, alternative implementations of an API can be loaded using shared libraries (ie. `.dll`, `.so`). Runtime options can be equally flexible, as functions can be called from a table and replaced dynamically.

In addition to providing greater flexibility in assembling software components, the API-based model attempts to address code readability—even within performance critical, low-level parts of the system, where macros are most commonly encountered. Figures 2.3 and 2.4 show this in the case of redefining macros for semaphore support, which provides an abstraction for controlling access to a shared resource across processes. The implementation of this support differs between the Linux and Windows operating systems, respectively; if the `LINUX` flag is set the `SEM_CREATE` macro is defined, otherwise it is left undefined.

2.3.1.1 Artefact Instances

Even in the small portion of Harmony's code base shown in Figures 2.3 and 2.4 we identify instances of three of the Rupture artefacts: *source*, *configuration* and *platform*.

The *source* artefact encompasses the portions of the two header files shown above. While it may seem that the source artefact would subsume all parts of the system, things like input files, makefiles and logging files we do not classify as source but still need to be considered.

```
1  /* SEM_CREATE */
2  #if defined(LINUX)
3   #define SEM_CREATE(initValue)
4     thread_malloc(NULL, sizeof(OSSEMAPHORE))
5  #else
6   #define SEM_CREATE(initValue)
7  #endif
8  /* SEM_INIT */
9  #if defined(LINUX)
10  #define SEM_INIT (sm, pshrd, inval)
11    (sem_init((sem_t*)sm, pshrd, inval))
12 #else
13  #define SEM_INIT(sm,pshrd,inval)
14 #endif
```

Figure 2.3: Macros usage for Linux in Harmony in `linux/thrdsup.h`.

```
1  /* SEM_CREATE */
2  /* Arbitrary maximum count */
3  #define SEM_CREATE(inval)
4  CreateSemaphore(NULL,inval,2028,NULL)
5  /* SEM_INIT */
6  #define SEM_INIT(sm,pshrd,inval)
7              (sm != NULL) ? 0: -1
```

Figure 2.4: Macros usage for Windows in Harmony in `windows/thrdsup.h`.

The *configuration* artefact demonstrated here leverages the C preprocessor for conditional inclusion in the Linux version (Figure 2.3, lines 2, 5, 7, 9, 12, 14) of semaphore support in the form of macros (Figure 2.3, lines 3, 4, 6, 10, 11, 13). Conditional compilation is a highly efficient method for tuning a master code base for multiple versions of a program in which different performance, size or functionality characteristics must be constructed. However, this fine-grained control over code placement associated with configuration and access to program state in the form of local variables and arguments makes it challenging for a developer to comprehensively manage source relevant to a specific build.

A key portion of the *configuration* artefact related to these code blocks, but not explicitly shown, is the selection of either the `Windows/threadsup.h` or `Linux/threadsup.h` file. This would be performed in a configuration file separate from the source listings shown here.

The *platform* artefact is demonstrated here in the coarse-grained directory struc-

ture breakdown between Windows and Linux. In addition to the coarse-grained selection of an operating system, more fine-grained platform decisions initialize semaphore support (Figure 2.3, lines 3, 4, 6, 10, 11, 13). The split source directory layout does provide a physical separation of Windows and Linux related code, but this structure does introduce redundancy across files which can introduce maintenance challenges.

While a tangible *design* artefact is not shown here, a key design principle used in the Harmony Portability Library is the isolation of platform-specific behaviour (ie. opening a file or a socket) behind a well-defined, platform-agnostic, API boundary. The API-based design keeps implementation and configuration options across architectures separate as demonstrated with the redefinition of semaphore support across two operating systems (Figures 2.3 and 2.4) showing the implementation of this design.

2.3.1.2 Artefact Relationships

In Section 2.3.1.1 the identification of individual artefacts as they play out within Harmony begins to demonstrate the intricate nature of dependencies between artefacts. Within just the few lines of code from Harmony we can see the causal relationship between *source*, *configuration* and *platform* artefacts. That is, the build of Harmony for a specific *platform* requires changes within *source* files and to *configuration* settings. For example, the decision as to whether a Windows or Linux *platform* will be supported is done within *configuration* files, this is a change that directly impacts which *source* will be included (which `.h` file). *Source* and *configuration* artefacts overlap in terms of the value of configuration flags set at a granularity of lines-of-code.

In the Linux build, the semaphore support has a defined functionality if the `LINUX` flag is set, otherwise it is empty (as indicated by the `#else` in Figure 2.3). In the Windows build, the macros have only one definition (Figure 2.4). This means that, in the case of `Linux`, the developer not only must know which header (`.h`) files are included in the build, but also how the `LINUX` flag controls the implementation of these macros within the *source*. It is thus critical for developers to be able to quickly determine the implications of a change to one artefact, even though not explicit, on all other artefacts.

Again, *design* artefacts are shown here, but we know from documentation that the goal is to keep the system infrastructure modular. The dominant decomposition of this modularity is dictated by the separation of platform details from source. As the

system continues to grow and take on new configurations to support more platforms it must continue to reflect this existing design goal, or if this is no longer possible, changes must be made to the *design* itself.

These examples demonstrate the intertwined nature of the relationships between changes to *platform*, *configuration*, *source* and *design* that impact dynamic characteristics of the system. Table 2.4 summarizes the results of the above analysis. examples of changes to *platform* that force additional changes to *configuration* and *design* as well changes to *configuration* that are within the *source* were identified in this analysis.

Table 2.4: Causal relationships between artefacts in Harmony.

causal artefact	impacted artefact
platform	configuration design
configuration	source

2.3.2 NAS Parallel Benchmarks (NPBs)

The NAS Parallel Benchmark (NPB) suite (NASA Advanced Supercomputing Division, 1999) provides a collection of eight programs designed to exercise underlying parallel architectures in terms of resource utilization and overall performance. Collectively, they mimic essential computation and data movement characteristics that typically create resource contention and utilization challenges in parallel systems.

Within this benchmark suite, there are three implementations of the eight programs: one serial implementation and two parallelized implementations, each of which is an incremental alteration of the serial implementation. The parallel implementations leverage the parallel support of OpenMP and MPI libraries with heavy reliance on compile time configuration tuning for architecture and input specific performance gains. Figures 2.5 and 2.6 show the use of OpenMP in the parallelization of benchmarks, `dc.c` and `is.c` respectively. Table 2.5 provides a summary of the configuration settings across nine different architectures for the OpenMP implementations of the NPBs alone.

```
1  #ifdef _OPENMP
2   adcpp->nTasks=omp_get_max_threads();
3   fprintf(stdout,``\nNumber of available threads:
4           %d\n", adcpp->nTasks);
5   if (adcpp->nTasks > MAX_NUMBER_OF_TASKS) {
6      adcpp->nTasks = MAX_NUMBER_OF_TASKS;
7      fprintf(stdout,``Warning: Maximum number of
8              tasks reached: %d\n", adcpp->nTasks);
9   }
10  #pragma omp parallel shared(pvstp) private(itsk)
11 #endif
12  {
13  double tm0=0;
14  int itimer=0;
15  ADC_VIEW_CNTL *adccntlp;
16 #ifdef _OPENMP
17  itsk=omp_get_thread_num();
18 #endif
```

Figure 2.5: Example of combined pragma and flag usage in `dc.c` of NPBs

```
1  #pragma omp parallel private(x,s,i,k)
2    {
3        INT_TYPE k1, k2;
4        double an = a;
5        int myid, num_procs;
6        INT_TYPE mq;
7  #ifdef _OPENMP
8        myid = omp_get_thread_num();
9        num_procs = omp_get_num_threads();
10 #else
11       myid = 0;
12   num_procs = 1;
13 #endif
```

Figure 2.6: Example of combined pragma and flag usage in `ic.c` of NPBs

2.3.2.1 Artefact Instances

From the NPB Figures 2.5 and 2.6 and Table 2.5 we extract instances of five of the *Rupture* artefacts: *source*, *platform*, *configuration*, *profiling* and *workload*.

Table 2.5: Configuration file settings for OpenMP across nine architectures in the NPB suite

Flag Type	ia64	ibm	ibm64	omni	pgi	sgi	sgi64	sun	sun64
Parallel C									
CC	ecc	xlc_r	xlf_r -q64	ompcc	pgcc	cc	cc -64	cc	cc
CFLAGS	-O3 -openmp	-O3 -qsmp=omp	-O3 -qsmp=omp	-xO4 -fast	-O3	-O3 -mp	-O3 -mp	-fast -xopenmp	-fast -xopenmp -xarch=native64
C_INC	null	null	null	null	null	null	null	null	null
CLINK	$(CC)	$(CC)	$(CC)	$(CC)	$(CC)	$(CC)	$(CC)	$(CC)	$(CC)
CLINKFLAGS	-O3 -openmp	-O3 -qsmp=omp	-O3 -qsmp=omp	-xO4 -fast	-O3	-O3 -mp	-O3 -mp	-fast -xopenmp	-fast -xopenmp -xarch=native64
C_LIB	null	null	null	null	null	null	null	null	Null
Utilities C									
UCC	ecc	cc	cc	cc	cc	cc	cc -64	cc	cc
BINDIR	../bin	../bin	../bin	../bin	../bin	../bin	../bin	../bin	../bin
RAND	randi8	randi8	randi8	randi8	randi8	randi8	randdp	randi8	Randi8
WTIME	wtime.c	wtime.c	wtime.c	wtime.c	wtime.c	wtime.c	wtime_sgi_64.c	wtime.c	wtime.c
MACHINE	n/a	-DIBM	-DIBM	n/a	n/a	n/a	n/a	n/a	n/a

The *source* artefact encompasses the two listings in Figures 2.5 and 2.6 but does not include the configuration settings shown in Table 2.5. These configuration options were gathered from nine separate configuration files within the NPB source download.

A variation in *platforms* is supported at the coarse-grained level of library support of OpenMP and MPI, with implementations separated through the package or directory structure. This analysis focuses on the OpenMP implementation as MPI targets a very different architecture. OpenMP is used more for shared memory programming whereas MPI is typically better for distributed systems. The column headings of Table 2.5 demonstrate the fine-grained *platform* options in the NPBs, with nine architectures being supported for OpenMP alone.

Configuration is a large part of the NPBs. Localized control of the *configuration* is attempted with a template `Makefile` in which configuration options/flags can be tuned for a specific build. For example, Table 2.5 demonstrates the kinds of flags

and settings that are involved in configuring one instance of this set of the OpenMP benchmarks across the nine architectures currently supported. More fine-grained *configuration* at the level of source code leverages the C-preprocessor to introduce OpenMP support to select the number of threads (Figure 2.5, lines 1-11,16-18 and Figure 2.6, lines 7-13).

Profiling in the NPB uses several strategies, including a mix of print statements (Figure 2.5, lines 7-8) and conditional compilation. The conditional inclusion of timing is based on the presence of an empty dummy file (`timer.flag`) in the current working directory which will create profiling artefacts embedded in source as:

```
1 #ifdef  TIMING_ENABLED
2       timer_stop( 2 );
3       timer_start( 3 );
4 #endif
```

Though not an automatic feature of most dynamic characteristics, *profiling* is also used to track configuration information during execution, augmenting output files with system characteristics. The following comment is associated with the implementation of this NPB *profiling* infrastructure within `sys/setparams.c`:
`This is a gross hack to allow the benchmarks to print out how they were compiled.`

Workload is varied through input files designed to benchmark underlying architectures with a wide range of sizes and types. The benchmark results shown in Figure 2.1 demonstrate this variability in *Class A* and *Class B* inputs, which only differ in the size of their principal arrays.

2.3.2.2 Artefact Relationships

The summary of individual artefacts in Section 2.3.2.1 in the NPBs highlights the significance of *configuration* artefacts in these kinds of highly customizable software systems. Changes to *configuration* are impacted by changes to *platform*, *profiling* and *workload*.

Specifically, Table 2.5 demonstrates the number of changes to *configuration* that must occur when the *platform* changes. Additionally, more fine-grained *configuration* changes using the `_OPENMP` flag are necessary to introduce OpenMP *platform*- specific changes at the level of *source*. Figure 2.5 shows the OpenMP `#pragma` nested within a conditional `#ifdef _OPENMP`, whereas Figure 2.6 shows the `#pragma` outside this

configuration artefact and as part of the *source*; whether it runs or not depends on the runtime platform: that is, if OpenMP is installed or not.

Profiling and *configuration* are tightly tied in this code base. The inclusion of the `timer` is dependent on the value of the `TIMING_ENABLED` flag and the *configuration* tracking is dependent on the implementation of the *profiling*. Both of these artefact instances reside within the *source*, bringing a third artefact into this interdependency.

This highly configurable system also highlights the dynamic impact of a change in *workload*, giving optimization tips for *configuration*. Specifically, the `README` for the OpenMP implementation specifies that 'an increase in input size *may* require a larger stack size' and gives the following suggestions for setting environment variables (configuration artefacts) for two specific architectures:

```
SGI Altix Intel compiler:  setenv KMP_STACKSIZE 50m

SGI Origin3000:  setenv MP_SLAVE_STACKSIZE 50000000
```

These examples demonstrate the intertwined nature of the relationships between changes to *platform*, *configuration*, *source*, *profiling* and *workload* that impact the *dynamic* characteristics of the system. Table 2.6 summarizes the results of this analysis. Identified in this analysis are examples of changes to *platform*, *workload* and *profiling* that force additional changes to *configuration*. Again, changes to both *configuration* and *profiling* are made within the implementation and therefore explicitly change the *source* artefact.

Table 2.6: Causal relationships between artefacts in NPBs

causal artefact	impacted artefact
platform	configuration
workload	configuration
profiling	configuration source
configuration	source

2.3.3 OmpSCR

The OpenMP Source Code Repository (OmpSCR) (Dorta *et al.*, 2005) is an infrastructure for benchmarks comprised of C, C++ and Fortran programs parallelized using OpenMP. These programs range from simple calculations to real scientific problems and are intended to both introduce OpenMP constructs and expose compiler

weaknesses. These benchmarks provide further evidence that the manifestation of *platform*, *configuration* and *profiling* artefacts are consistent with those found in Harmony and the NPBs.

We draw from a small example program from the OmpSCR benchmark suite to provide concrete examples of the causal relationships and crosscutting nature of the artefacts described above. Figures 2.7, 2.8 and 2.9 provide listings of the functions that make up the *molecular dynamic* application in the `c_md.c` file, within the OmpSCR benchmarks. Figures 2.8 and 2.9 show the functions that perform the computation in this application. Each function has a fully parallel region (Figures 2.8, lines 17-36 and Figure 2.9, lines 13-19) parallelized with OpenMP.

2.3.3.1 Artefact Instances

From Figures 2.7, 2.8 and 2.9, which comprise the *molecular dynamic* application, we extract instances of five of the Rupture artefacts: *source*, *platform*, *configuration*, *profiling* and *workload*. Once again, each of these three listings from OmpSCR can be considered *source* artefacts from a course-grained perspective, but a line-by-line analysis demonstrates the fine-grained, intricate nature of the other artefacts.

An example of *profiling* set up is shown in the `main` method in Figure 2.7 with a timer (lines 5,7) and the use of `printf` for logging of results and possible accuracy monitoring (lines 29-32).

In terms of *platform*, since OmpSCR is an OpenMP specific benchmark, it does not support any other parallel mechanisms. The code does not leverage the C-preprocessor to conditionally introduce OpenMP constructs, it assumes OpenMP is part of the installed platform.

Much of the *configuration* associated with parallelization is left to the underlying OpenMP libraries, delegated-to by the `#pragma` lines that precede a parallelizeable loop (Figure 2.8, line 15 and Figure 2.9, line 11). OpenMP uses a library routine to establish the maximum number of threads, shown in the following line from OmpSCR `c_md.c`:

```
NUMTHREADS = omp_get_max_threads();
```

which will use an environment variable, `OMP_THREAD_LIMIT`, to control the thread population size. Fine-grained *configuration* leveraging the C-preprocessor is also present in these listings (Figure 2.7, line 29,32).

The *workload* in terms of the number of parts to be processed and the number of

```
1  int main (int argc, char **argv) {
2    /* simulation parameters */
3    ...
4    int NUMTHREADS;
5    double total_time;
6    char *PARAM_NAMES[NUM_ARGS] = {``Nparts", "Nsteps"};
7    char *TIMERS_NAMES[NUM_TIMERS] = {``Total_time" };
8    char *DEFAULT_VALUES[NUM_ARGS] = {``8192", ''10"};
9
10   NUMTHREADS = omp_get_max_threads();
11   OSCR_init (NUMTHREADS, ``Molecular dynamic simulation",
12     ``Use md <Nparts> <Nsteps>", NUM_ARGS, PARAM_NAMES, DEFAULT_VALUES,
13     NUM_TIMERS, NUM_TIMERS, TIMERS_NAMES, argc, argv);
14
15   NPARTS = OSCR_getarg_int(1);
16   NSTEPS = OSCR_getarg_int(2);
17
18   ...
19   NUMTHREADS = omp_get_max_threads();
20
21   ...
22   /* compute the forces and energies */
23   compute(NPARTS, NDIM, position, velocity, mass, force, ...);
24   E0 = potential + kinetic;
25
26   /* This is the main time stepping loop */
27   for (i = 0; i < NSTEPS; i++) {
28     compute(NPARTS, NDIM, position, velocity, mass, force, ...);
29 #if 0
30     printf(``%17.9e %17.9e %17.9e\n", potential, kinetic,
31            (potential + kinetic - E0) / E0);
32 #endif
33     update(NPARTS, NDIM, position, velocity, force, accel, mass, dt);
34   }
35   ...
36   return 0;
37 }
```

Figure 2.7: The `Main` function in OmpSCR's molecular dynamic application, `md.c`.

steps (Figure 2.7, line 6) of this particular application has a default setting (line 8) which can be overridden by command line arguments (lines 15,16).

2.3.3.2 Artefact Relationships

As demonstrated in Harmony and the NPBs, *configuration* is intertwined with multiple artefacts within the OmpSCR benchmarks including *profiling*, *workload*, *source* and *platform*. *Profiling* is turned on and off with preprocessor directives controlled

```
1  /* ------------------------------------------------------------------
2     Compute the forces and energies, given positions, masses,
3     and velocities
4   * ------------------------------------------------------------------ */
5  void compute(int np, int nd, vnd_t *pos, vnd_t *vel,
6               double mass, vnd_t *f, double *pot_p, double *kin_p) {
7    int i, j, k;
8    vnd_t rij;
9    double d;
10   double pot, kin;
11
12   pot = 0.0;
13   kin = 0.0;
14   /* The computation of forces and energies is fully parallel. */
15 #pragma omp parallel for default(shared) private(i, j, k, rij, d)
16           reduction(+ : pot, kin)
17   for (i = 0; i < np; i++) {
18     /* compute potential energy and forces */
19     for (j = 0; j < nd; j++)
20       f[i][j] = 0.0;
21     for (j = 0; j < np; j++) {
22       if (i != j) {
23         d = dist(nd, pos[i], pos[j], rij);
24         /* attribute half of the potential energy to particle 'j' */
25         pot = pot + 0.5 * v(d);
26         for (k = 0; k < nd; k++) {
27           f[i][k] = f[i][k] - rij[k]* dv(d) /d;
28         }
29       }
30     }
31     /* compute kinetic energy */
32     kin = kin + dot_prod(nd, vel[i], vel[j]);
33   }
34   kin = kin * 0.5 * mass;
35   *pot_p = pot;
36   *kin_p = kin;
37 }
```

Figure 2.8: The `compute` function in OmpSCR's molecular dynamic application, `md.c`.

by a *configuration* flag within the *source* (Figure 2.7, line 29-32). Also, within *source* files, the setting of fields such as `NUMTHREADS`, `NPARTS` and `NSTEPS` (Figure 2.7, lines 10,15,16,19) is a change in *configuration* but in turn also impacts the *workload*.

Configuration is highly dependent on *platform* settings such as whether OpenMP is installed to enable the use of mechanism-specific operations (Figure 2.7, lines 10,19). Changes to any one of these static artefacts will have a dynamic impact in terms of performance and/or resource utilization; this may require causal changes to cascade

```
1  /* ------------------------------------------------------------------------
2     Perform the time integration, using a velocity Verlet algorithm
3   * ------------------------------------------------------------------------ */
4  void update(int np, int nd, vnd_t *pos, vnd_t *vel, vnd_t *f,
5              vnd_t *a, double mass, double dt) {
6    int i, j;
7    double rmass;
8
9    rmass = 1.0/mass;
10   /* The time integration is fully parallel */
11 #pragma omp parallel for default(shared) private(i, j)
12           firstprivate(rmass, dt)
13   for (i = 0; i < np; i++) {
14     for (j = 0; j < nd; j++) {
15       pos[i][j] = pos[i][j] + vel[i][j]*dt + 0.5*dt*dt*a[i][j];
16       vel[i][j] = vel[i][j] + 0.5*dt*(f[i][j]*rmass + a[i][j]);
17       a[i][j] = f[i][j]*rmass;
18     }
19   }
20 }
```

Figure 2.9: The `update` function in OmpSCR's molecular dynamic application, `md.c`.

again through these related artefacts.

To demonstrate the challenges associated with tracking these overlapping artefacts, consider an example of the dependency between *source* and *configuration* artefacts that has dynamic impact on the number of threads executing. Although the value returned by `omp_get_max_threads()` in this example returns the value of the `OMP_THREAD_LIMIT` environment variable, there are three other ways to achieve this same result: 1) a corresponding OpenMP routine to set the number of threads within the *source*, 2) a *configuration* `#pragma` to set the number of threads for each parallel region and 3) the `OMP_NUM_THREADS` environment variable in the *configuration* which establishes an initial number of threads. At any point in the program, the actual number of threads working depends on the order of execution of the `set` routines versus the extent of *configuration* occurring in the local `#pragma`s, or in the absence of both, the value of the environment variable set within the *configuration* artefact.

These examples demonstrate the causal relationships between changes to *platform*, *configuration*, *source*, *profiling* and *workload* that impact dynamic characteristics of the software. Table 2.7 summarizes the results of this analysis. Identified in this analysis, are examples of changes to *platform* and *profiling* that force additional changes to *configuration* as well as changes to *platform* that force changes to *source*.

Changes to both *configuration*, *workload* and *profiling* are made within *source* and consequently change the *source* artefact.

Table 2.7: Causal relationships between artefacts in OmpSCR

causal artefact	impacted artefact
platform	configuration source
profiling	configuration source
configuration	source workload
workload	source

2.4 Summary

In order to address objectives O1 and O2:

- **O1.** Determine what is easy/hard about parallel software development.
- **O2.** Discover artefacts key to the parallel software development process.

this chapter provides an analysis of parallel code bases, in terms of existing software development processes, to establish the challenges that are introduced and amplified within the parallel domain. It is this analysis that allows us to identify what information is valuable and relevant for a parallel developer. In addition to the identification of critical information or key artefacts, this work identifies relationships between these artefacts, establishing the requirements for the Rupture model. The Rupture model provides a proof of concept process, based on the artefacts and relationships identified as critical to parallel software development.

The contribution of this chapter is the identification of key artefacts and relationships between artefacts that are necessary to track and reason about within the parallel software development process. From this contribution we propose Rupture as a possible model to support this process for parallel applications.

The dynamic impact of configuration and platform optimizations was identified as an integral part of tracking parallel benchmarks. Drawing from the contribution

requirements set out by an existing benchmark suite, we established a set of critical artefacts of which developers must be cognisant within the parallel development process. Examples of these artefacts and two types of causal relationships that exist between them are proposed. The identification of these artefacts, combined with the importance of performance as a non-functional requirement for parallel applications, demonstrates the need for parallel specific workflows that include feedback from dynamic results back into static artefacts. This workflow is realized within the Rupture model.

The artefacts of the Rupture model are evaluated in terms of their existence in three different parallel software projects. Though the analysis hinges on examples from low-level code bases with limited variation of parallel mechanisms, we believe these results generalize to other systems, applications and languages.

Specifically, this chapter serves to carve out the manifestations of and relationships between the artefacts in the Rupture model, and the precise elements that can be changed, tweaked and tracked in the parallelization of code bases.

In summary, the contributions of this chapter are:

- Documented the process through which a parallel programmer gains feedback during software development.
- Determined how existing software development lifecycle models are typically structured.
- Documented opportunities to collect valuable information within the development process.
- Used the opportunities to identify artefacts and causal relationships between artefacts.
- Evaluated the artefacts and relationships in the context of existing code bases.

The following chapter investigates and evaluates systematic approaches for a developer to compare and analyse examples of these critical artefacts in the form of design patterns.

Chapter 3

RIPL: A Systematic Methodology for Parallel Pattern Analysis

In this chapter we begin our investigation of ways to track and understand key artefacts within parallel software development. We begin by investigating an essential artefact of the design phase, the *design pattern*. Design patterns, which describe an abstract solution to a frequently occurring software problem, were made common in the object-oriented domain by the Gang of Four (Gamma *et al.*, 1995) and are now a widely accepted design artefact in many paradigms.

Targeting research objective **O3** of the thesis, in this chapter we use design patterns as our first artefact in the investigation of approaches to support analysis and comparison of parallel software artefacts. This work has been vetted within the pervasive (Gibbs and Coady, 2009) and parallel pattern (Gibbs and Coady, 2010b) communities. In support of this objective, this chapter specifically addresses the following research question:

> *What kind of systematic approaches could help in the analysis and comparison of parallel patterns to support the selection process and identify where existing parallel patterns fall short?*

3.1 Introduction

Patterns for Parallel Programming (Mattson *et al.*, 2004), written by experts in parallel development from academia and industry, presents a collection of parallel patterns in the form of a pattern language. The patterns are grouped by design spaces at

three levels of generality: *finding concurrency*, *algorithm structure* and *supporting structure*. The intent is to provide a guide through the pattern selection process with a methodological approach described in chapter entitled *Finding Concurrency*. This chapter focuses on support for overall application design providing insights in terms of which algorithms and data structures work to augment the *Algorithm* and *Supporting Structure* chapters which provide the details of these patterns.

Existing pattern languages successfully describe relationships in small collections of patterns, but this approach lacks a systematic process that will scale to a growing catalogue of patterns. Some parallel pattern writers are attempting to address this issue of pattern selection by collecting small families of patterns and explicitly representing the relationships between patterns as a focal point within the pattern language.

For example, in a pattern language containing four sorting patterns (Kale, 2010), the sections of the pattern write-ups are consolidated to support comparison across patterns. The authors start with an overarching *forces* section that identifies a global set of *forces* relevant to all four sorting patterns. Following the *forces* section, each individual pattern write-up includes both a qualitative and quantitative resolution of the set of *forces*. An overview of the resolution of each *force* by all the patterns is provided in table form in Table 3.1. This consolidated approach provides support for pattern comparison for the purpose of pattern selection.

We propose the development of a localized, scalable structure which lists the tradeoffs made within the patterns of a given pattern language. This approach is intended to allow developers to modularly view the similarities and differences between patterns in terms of design and implementation- level constraints.

The remainder of this chapter is organized as follows: a methodology for pattern comparison motivated by existing work in the parallel domain is proposed (Section 3.2), followed by two case studies evaluating this approach. We then demonstrate success in the pervasive domain case study (Section 3.3) and promise further challenges in the parallel domain (Section 3.4) with a summary reflecting on contributions and lessons learned (Section 3.5).

3.2 Proposal

Our proposed solution, *Relationship Initiated Pattern Language (RIPL)*, focuses on relationships between *forces* across patterns. From this perspective, we can align goals

Table 3.1: Summary of forces assessment for parallel sorting solutions (Kale, 2010)

Algorithm	Quicksort	Sample Sort	Histogram Sort	Radix Sort
Load Balance	Good	Good	Excellent	Good
Data Movement	Good	Excellent	Excellent	Moderate
Communication Latency	Excellent	Moderate	Moderate	Moderate
Additional Bandwidth	Good	Good	Good	Good
Exploitation of Initial Distribution	Good	Moderate	Good	Poor
Overlapping Communication and Computation	Moderate	Good	Excellent	Moderate

between patterns and support comparison in terms of known tradeoffs. Furthermore, domains with shared constraints can be more easily combined. As multi-core architectures continue to become a commodity, the future arguably holds convergence of parallel and pervasive worlds. In order to effectively reason across this design space, we need a common pattern language that can allow us to examine tradeoffs and coordinate compositions, consistent with an overall design methodology.

To establish the relationships between tradeoffs of different patterns we begin by considering patterns in terms of their *forces*—that is, the goals they are trying to achieve (*dominant force*), and the opposing constraints (*conflicting forces*) they encounter. This breakdown can be directly obtained through each individual pattern description. For example, extracting *forces* from descriptions of four sample patterns from within Berkeley's parallel pattern language, *Task Parallelism* (Chong, 2009b), *Data Parallelism* (Mattson *et al.*, 2004), *Discrete Event* (Andrade, 2009), and *Speculative Execution* (Chong, 2009a), can be summarized as in Table 3.2. One thing that becomes immediately apparent in the format of Table 3.2 is that the *forces* enumerated in the forces column are: (1) tractable, and many of them are shared between patterns and (2) not limited to the parallel domain. Leveraging these observations, we developed RIPL as a *force-centric* pattern language designed to scale across multiple

domains including the parallel domain.

Table 3.2: Parallel patterns and the *forces* that impact them

Pattern	Forces
Task Parallelism	data dependencies ordering constraints task granularity load balancing
Data Parallelism	data dependencies task granularity load balancing
Discrete Event	data dependencies ordering constraints
Speculation	conflict resolution task granularity

RIPL organizes patterns according to core categorical themes: *task*, *communication* and *data*, augmented with a crosscutting, *force-centric* theme that enables a view of the relationships between patterns based on the constraints they encounter illustrated in an unpopulated view in Figure 3.1. The horizontal pattern layers of RIPL are inspired by the Berkeley pattern language structure; the vertical force column is motivated by the tradeoff information conveyed within the *forces* section of each pattern. When populated with a set of patterns, this organization will allow developers to simultaneously consider top-down goals in terms of categories of patterns from which to choose from, and bottom-up constraints in terms of the *forces* to be considered; this provides a holistic perspective of a design space.

We have further coupled RIPL with a systematic methodology for the bottom-up reasoning about design decisions based on key objectives across patterns. Specifically, one *force* can be considered as having a priority in a given context, or as a *dominant force*, and the corresponding *conflicting forces* can be reasoned about across patterns in the language.

Like individual patterns, *forces* can also be categorized in terms of the core issues of *task*, *communication* and *data*, but with an additional group for *non-functional forces*, as shown in Figure 3.2. Design patterns typically provide a solution in terms of some functionality; *non-functional* issues such as performance while in many cases are addressed within a given pattern, are not the main focus. However, in the parallel

Forces	Task Patterns
	Communication Patterns
	Data Patterns

Figure 3.1: The proposed structure of RIPL (Relationship Initiated Pattern Language)

domain it is essential to consider environmental and architectural *forces*, as they are fundamental to achieve acceptable resource utilization and reap performance benefits.

Core Issue	Dominant Forces	Conflicting Forces			
		p_1	p_2	...	p_n
Non-Functional	F_1	*	*		
	F_2		*		
Task					
		*		*	
Communication				*	*
Data		*	*		
	F_n				

Figure 3.2: Explicit representation of tradeoffs in RIPL in terms of *forces*

Essentially, for each design decision that addresses a *dominant force* (F_1, F_2, ...F_n), each pattern (p_1, p_2, ...p_n) will reveal the decision points that constitute tradeoffs. These decision points are in terms of *conflicting forces*, generally represented in Figure 3.2 as a '*'; are the *forces* that must be sacrificed in order to satisfy the *dominant force* (column 2). It is through this structure that RIPL promotes the systematic comparison of tradeoffs across patterns in terms of the *forces* they address and the *forces* that are conceded in return. As indicated by the shaded cells, conflicts not identified by a given set of pattern definitions is inevitable.

3.3 RIPL Case Study: Pervasive Patterns

In order to evaluate RIPL and its methodology we apply it in a case study to two existing pervasive pattern languages: *TinyOS* (Gay *et al.*, 2007) and *Agent* (Deugo *et al.*, 2001). The pervasive domain was selected as an initial case study because of its similarity to the parallel domain and the availability of already established pervasive design patterns. Patterns in the pervasive domain deal with issues centred around the need to organize multiple agents to work together, much like parallel design patterns which must organize multiple compute units to work together.

Using these patterns, we populate the proposed pattern language structure shown in Figure 3.1 and outline the tradeoffs identified by the pattern authors. From these tradeoffs we extrapolate a list of *forces* specific to the pervasive domain, which are used to further populate RIPL and construct a list of *forces* to be used in the tradeoff comparison illustrated in Figure 3.2.

3.3.1 Pattern Overview

The *TinyOS* pattern language is a collection of implementation-specific patterns influenced by the *TinyOS* environment (TinyOS, 2009) drawing code examples in the domain-specific NesC language (Gay *et al.*, 2003). The *Agent* pattern language is a collection of generalized solutions for coordination of agents to work together in pervasive environments. These patterns cover a spectrum of coordination strategies, each addressing various *forces* with a spectrum of tradeoffs.

3.3.1.1 TinyOS Patterns

The *TinyOS* pattern language consists of five patterns that provide guidance in coordination of services within and across agents or motes: *Service Instance*, *Placeholder*, *Dispatcher*, *Keyset* and *Keymap*. The *Service Instance* pattern promotes multiple instances of a single service to be used by multiple collaborators. The *Dispatcher* and *Placeholder* patterns support service selection from various implementations of a given interface, where the selection for *Dispatcher* is done at runtime and *Placeholder* at compile time. *Keyset* and *Keymap* are lower level patterns for managing state and data communication across motes. The *Keymap* pattern provides a mapping between two *Keysets* within different *Service Instances*, providing a good example of pattern composition.

3.3.1.2 Agent Patterns

The *Agent* pattern language consists of five patterns that coordinate agent communication: *Blackboard*, *Meeting*, *Market Maker*, *Master-Worker* and *Negotiator*. *Blackboard*, *Meeting* and *Market Maker* all introduce an intermediary to coordinate communication between agents. The *Master-Worker* and *Negotiator* patterns provide alternate strategies for coordinating work across agents; the *Master-Worker* pattern is described as being vertical coordination across one higher level agent and one or more lower level agents, whereas the *Negotiator* pattern is described as horizontal coordination where the agents are equal participants.

3.3.2 Extrapolating Tradeoffs

Tables 3.3 and 3.4 overview tradeoffs identified by *TinyOS* and *Agent* pattern authors, respectively. These tradeoffs are broken down in terms of the core layers in RIPL: *non-functional*, *task*, *communication* and *data* to categorize the *forces* involved in each tradeoff. This subsection provides a summary of how these tradeoffs were identified.

Table 3.3: Tradeoffs in TinyOS Patterns

	Service Instance (SI)	Placeholder (PH)	Dispatcher (DP)	Keyset (KS)	Keymap (KM)
Non-Functional	memory usage vs flexibility	memory usage vs flexibility			memory usage vs coordination complexity
Task	coordination complexity vs flexibility	flexibility vs coordination complexity	flexibility vs coordination complexity		
Communication		communication protocol vs communication overhead	communication protocol vs communication overhead	communication protocol vs coordination complexity	communication protocol vs coordination complexity
Data				identifier management vs coordination complexity	identifier management vs coordination complexity

Table 3.4: Tradeoffs in Agent Patterns

	Blackboard (BB)	Meeting (MT)	Market Maker (MM)	Master-Worker (MW)	Negotiator (NG)
Non-Functional	mobility vs communication protocol	mobility vs communication protocol	mobility vs communication protocol		
	transience vs communication protocol	transience vs communication protocol	transience vs communication protocol		
	mobility vs security	security vs coordination complexity			
	transience vs security				
Task	coordination complexity vs flexibility	flexibility vs coordination complexity	flexibility vs coordination complexity	load balancing coordination complexity	
Communication	communication overhead vs identifier management	communication overhead vs identifier management	communication overhead vs identifier management		action alignment vs communication overhead
	communication overhead vs communication protocol	communication overhead vs communication protocol	communication overhead vs coordination complexity	communication protocol vs coordination complexity	communication protocol vs coordination complexity
Data				data dependency vs coordination complexity	
				load balancing vs communication overhead	

3.3.2.1 TinyOS Tradeoffs

Looking at Table 3.3, we see *Service Instance*, *Placeholder* and *Keymap* are the only patterns that mention environmental, ***non-functional*** *forces* (Table 3.3, row 1). Specifically, the *Keymap* pattern looks to conserve memory space by accepting the overhead of coordinating local compact name spaces globally. *Service Instance* and *Placeholder* address the tradeoff of the *flexibility* of dynamic dispatch versus taking up less memory space with dispatch tables in compiled code.

Service *flexibility*, categorized here under the core issue of ***task*** *forces* (Table 3.3, row 2) is also opposed by the *coordination complexity* required to support it. *Service Instance*, *Placeholder* and *Dispatcher* each introduce increasing levels of *coordination complexity* for a range of *flexibility* ranging from very little in *Service Instance* to the dynamic variability in *Dispatcher*.

The tradeoffs associated with ***communication*** (Table 3.3, row 3) include the cost of *communication overhead* introduced by the more enhanced *communication protocol* of the *Placeholder* and *Dispatcher* patterns. The increased *coordination complexity* required for *Keyset* and *Keymap* gives these patterns an enhanced *communication protocol* for the required communication across agents.

This *identifier management* handled by *Keyset* and *Keymap* patterns falls into ***data*** (Table 3.3, row 4) due to the attention in these patterns to maintenance of state and data shared across agents. This *force* is directly impacted by the *coordination complexity* of organizing data across local and global services.

3.3.2.2 Agent Tradeoffs

The *Agent* patterns in Table 3.4 are more cognizant of the ***non-functional*** *forces* (Table 3.4, row 1), with *Blackboard*, *Meeting* and *Market Maker* all discussing pervasive environmental *forces* of agents changing locations, coming in and out of the system and the *security* associated with this *transience*. The simple *communication protocol* facilitated by an intermediary in these patterns address *forces* of *mobility* and *transience* by allowing agents to register/deregister dynamically. This solution introduces *security* issues in terms of which agents should be communicating. The *Meeting* pattern introduces extra measures in terms of *coordination complexity* to address these *security* issues.

The ***task*** related *forces* (Table 3.4, row 2) discussed in these patterns further weigh the coordination complexity versus the flexibility of the service. Similar to the

TinyOS patterns, an increasing amount of coordination is required across *Blackboard*, *Meeting* and *Match Maker* to gain *flexibility*. *Master-Worker* is a more *task-centric* pattern, trading off *coordination complexity* of task organization by the master to achieve *load balancing* across workers.

The focus of *Agent* patterns is ***communication*** across agents (Table 3.4, row 3). *Communication overhead* is mitigated by the simplicity of the *communication protocol* of the mediator style design of the *Blackboard*, *Meeting* and *Market Maker* patterns. This mediator style eliminates the need for *identifier management* and at the same time minimizes *communication overhead* with direct communication.

The *Blackboard* and *Meeting* patterns are limited to simple *communication protocols* in order to further minimize *communication overhead*. The physical locality of the *Meeting* pattern's mediator additionally reduces communication costs by eliminating *mobility* of the intermediary. The coordination logic of the *Meeting* and *Market Maker* patterns resides in a separate module, improving the *flexibility* of *communication protocols*, and mitigating the *coordination complexity* added with the *communication protocol* of *Market Maker*.

The *Master-Worker* and *Negotiator* patterns address *forces* associated with communication across agents, where the *Master* is concerned with *load balancing* across workers and the *Negotiator* pattern is concerned with *action alignment* of agents working together as equals. These more enhanced *communication protocols* require much more in the form of *coordination complexity*. In addition, *Master-Worker* keeps the coordination transparent to the client using the service. The *Negotiator* pattern requires each agent to be explicit about its intentions and all agents to agree before computation begins, which requires extensive coordination.

Data issues (Table 3.4, row 4) are not yet a large focus in pervasive systems. *Master-Worker* and *Negotiator* must manage the coordination/delegation of work across agents. This introduces *coordination complexity* in terms of managing data dependencies across tasks and *communication overhead* for *load balancing*. Both *data* and *task load balancing* are goals in pervasive environments; for this reason it is duplicated across the two categories.

3.3.3 Results

Given this analysis of ten pervasive patterns from two pattern languages, we can now to populate the proposed pattern structure as depicted in Figure 3.1 and the

force decision matrix originally of Figure 3.2. Figures 3.3 and 3.4 demonstrate the population of the dominant and crosscutting pattern language structure respectively, with a supporting description in the following two subsections. Again, these two perspectives work in concert to provide a top-down design view combined with a bottom-up view of design tradeoffs.

3.3.3.1 Populating the Pattern Language Structure

At the level of *Pervasive Task Patterns* in Figure 3.3, *Service Instance*, *Placeholder* and *Dispatcher* provide abstractions addressing the core issue of *task*. Additionally, the main tradeoff addressed by these patterns is between flexibility of service and the complexity associated with the coordination of this service.

The majority of the *Agent* patterns reside at the level of *Communication Patterns* in Figure 3.3 as their key goal is the coordination of communication. Specifically, *Blackboard*, *Meeting*, *Market Maker*, *Master-Worker* and *Negotiator* populate this layer of the pattern structure backed by their attention to *communication-centric forces* of *communication overhead* and *communication protocol*. Specifically, the *Negotiator* pattern supports explicit *action alignment* between agents and so requires extra communication before task collaboration even begins.

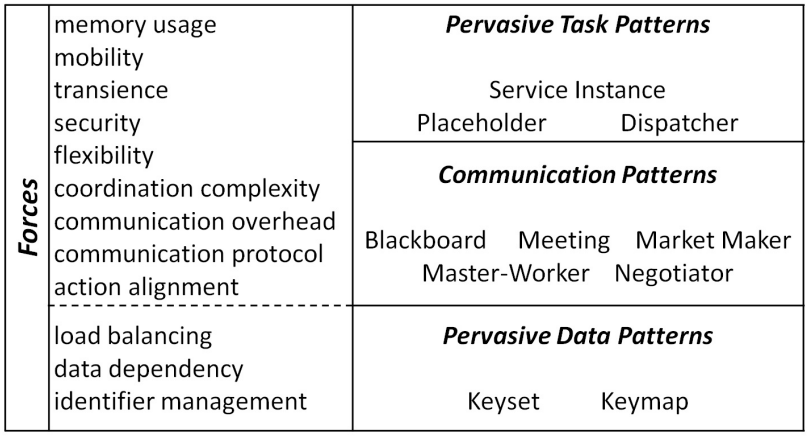

Figure 3.3: Example population of pervasive pattern language structure

Keyset and *Keymap* are the only patterns that are allocated to the *Pervasive Data Patterns* layer. These patterns, while facilitating communication, are largely focused on the maintenance and tracking of both state and data for a task. While these *forces* seem to be associated with coordination of communication, this coordination

is performed through *identifier management* by way of maintenance of state data.

The tradeoff analysis in Section 3.3.2 provides a core set of *forces* specific to the domain of pervasive systems. This set is neither exhaustive, nor complete but serves as a proof of concept. These core *forces* address *non-functional*, *task*, *communication* and *data* issues and include: *memory usage*, *mobility*, *transience*, *security*, *flexibility*, *coordination*, *complexity*, *communication overhead*, *communication protocol*, *action alignment*, *load balancing*, *data dependency* and *identifier management*. The *forces* themselves are loosely tied to one of the three focal points of *task*, *communication* and *data* but as the dotted lines between layers implies, the *forces* crosscut these boundaries and apply to more than one category.

	Dominant Forces	***Conflicting Forces***									
		BB	*MT*	*MM*	*MS*	*NG*	*SI*	*PH*	*DP*	*KS*	*KM*
Non- Functional	memory usage (MU)						FL	FL			CC
	mobility (MO)	SE, CP	CP	CP							
	transience (TR)	SE, CP	CP	CP							
	security (SE)		CC								
Task	flexibility (FL)		CC	CC				CC	CC		
	load balancing (LB)				CC						
	coordination complexity (CC)	FL					FL				
Communication	communication overhead (CO)	CP, IM	CP, IM	IM			CP				
	communication protocol (CP)			CC		CC		CO	CO	CC	CC
	action alignment (AA)					CO					
Data	data dependency (DD)				CC						
	identifier management (IM)									CC	CC
	load balancing (LB)				CO						

Figure 3.4: A population of the four categories of the force-centric pervasive pattern language.

3.3.3.2 Populating the Crosscutting Structure

The population of RIPL shown in Figure 3.4 comes directly from the tradeoffs in each pattern, where the *dominant forces* column is an enumeration of the *forces* encountered across all ten patterns. The *forces* are organized in terms of the four categories: *non-functional*, *task*, *communication* and *data*. As noted earlier, *load balancing* can refer to both a *task* or a *data force* and so is entered in both categories.

As an example, looking at RIPL from a column perspective, we can see that *Blackboard* addresses the *forces* of *mobility*, *transience*, *coordination complexity* and *communication overhead* as those cells are not grayed out. The actual values of those corresponding cells identify the *forces* that are compromised in the process of addressing the *dominant force*.

Specifically, *Blackboard* addresses *mobility* and *transience* by giving up *security* and enhanced *communication protocol* to achieve it. The *Blackboard* pattern minimizes *coordination complexity* in exchange for reduced *flexibility*. Finally, shown in the *communication overhead* row, *Blackboard* minimizes *communication overhead* by limiting the *communication protocol* and not providing *identifier management*.

The whole structure of RIPL is populated in this way from the tradeoffs summarized in Tables 3.3 and 3.4.

3.4 RIPL Case Study: Parallel Patterns

To evaluate RIPL in the parallel domain we applied its approach to two newly developing parallel pattern languages: *Scalable Sorting Patterns* and Berkeley's *Parallel Pattern Language (PPL)*. PPL is pioneering work in the area of design patterns for parallel programming and providing a test bed for the evaluation of RIPL. PPL itself is incomplete and continues to develop and evolve. The *Scalable Sorting Patterns* (Kale, 2010), a collection of three parallel sorting patterns not written in the classic GOF format, provides a different point on the spectrum for evaluation. The comparison-focused write up based on *forces* that this pattern language employs should align with the RIPL approach.

3.4.1 Pattern Overview

PPL is a growing collection of parallel specific patterns with contributions from various academic institutions including Berkeley and Illinois State. The Scalable Sorting

Patterns was a submission to the annual Workshop on Parallel Programming Patterns (Paraplop) (Johnson *et al.*, 2010). This selection provides a range of patterns from both a large, growing pattern language (PPL) and a smaller, solution-specific set of patterns (Scalable Sorting Patterns).

3.4.1.1 Parallel Programming Patterns

The working version of the merge of the Berkeley PPL and the *Pattern Language for Parallel Programming* (PLPP) is hosted on the Berkeley patterns wiki (Berkeley Parallel Computing Lab, 2010). In this case study we considered a snapshot of version 2.0 taken in April 2009 (Figure 1.8 in Chapter 1). At this point the pattern language structure consisted of four layers or levels: *Application Patterns* which contain *Structural* and *Computational Patterns* at layer one, supported by *Parallel Algorithm Strategy Patterns*, *Implementation Strategy Patterns* and *Concurrent Execution Patterns* at the underlying layers. With 57 patterns populating this structure across the four layers, many of the pattern descriptions have not been written up formally, but their place within the pattern language has been established.

In this case study we limit our view to the more populated sections of the *Structural* and *Algorithm Strategy* patterns. We analyze the eight patterns defined at the structural level and six at the algorithm level.

Agent & Repository, *Pipe & Filter*, *Process Control*, *Iterative Refinement*, *Event Based Implicit Invocation*, *Layered Systems*, *MapReduce* and *Model View Controller* all provide guidance for the structure of program and data together. Strategies range from a divide-and-conquer type of approach of *MapReduce* to the more event-driven structure of *Pipe & Filter*, *Process Control* and *Event Based Implicit Invocation*. Patterns such as *Agent & Repository*, *MapReduce* and *Event Based Implicit Invocation* focus more on distribution than the other patterns. This information is not explicit from the pattern language layout. That is, a developer must delve into each pattern individually to understand the subtle differences and similarities in tradeoffs.

3.4.1.2 Scalable Sorting Patterns

The *Scalable Sorting Pattern* (Kale, 2010) is comprised of three different sorting algorithms: *Sample Sort*, *Quick Sort* and *Radix Sort*; each is considered here as an individual pattern. These sorting patterns apply different algorithms to split up a data set to distribute the sorting across multiple compute units. Each algorithm provides

a different strategy for *distribution* and *load balancing* which has direct implications in terms of *memory usage*, *communication overhead* and *runtime complexity*.

3.4.2 Extrapolating Tradeoffs

Unlike the pervasive case study (Section 3.3), we were unable to extract an exact set of tradeoffs in the form of a *dominant force* and the *conflicting force* from each pattern in both the sorting and PPL patterns.

Table 3.7 overviews the tradeoffs in terms of *forces* that impact the patterns that make up the *Scalable Sorting Pattern*, whereas Tables 3.5 and 3.6 outline the *forces* that impact each of the structural and algorithm patterns of the PPL. The following two subsections provide an overview of how these *forces* and tradeoffs manifest themselves in the sorting and parallel patterns described above.

3.4.2.1 PPL Tradeoffs

Tables 3.5 and 3.6 outline an extracted list of *forces* from the current listing of patterns in PPL's *Structural* and *Parallel Algorithm Strategy* sections, respectively. Although the discussion style of many of these patterns' *forces* does not lend itself cleanly to the strict methodology proposed by RIPL, we make a best-effort attempt to itemize the *forces* for the 14 patterns in question. The shading across the two tables serves to highlight the *forces* that occur in both categories of patterns.

Table 3.5 shows a list of 25 *forces* mentioned in the 8 structural patterns sampled from the PPL, with an 'X' indicating the pattern each *force* impacts. In a first look at this representation of the *forces*, we can see that some appear to be pattern-specific while others impact multiple patterns. For example, *fault tolerance*, *customizability* and *portability* are mentioned only in *Model View Controller*, whereas *code complexity*, *throughput* and *load balancing* are *forces* common to multiple patterns.

Through the pattern analysis that generated this table, it appears that forces are not presented in a flat structure. There may actually be higher-level decisions, such as *distribution* versus *centralization*, that in turn dictate further *forces* separately, such as *communication overhead* and *load balancing*. It may be worthwhile to leverage this structure more systematically.

Table 3.6 provides an overview of the *forces* mentioned in the six *Parallel Algorithm Strategy* patterns of the PPL. This set of patterns had a smaller set of *forces*,

Table 3.5: Forces impacting the PPL structural patterns.

Forces	Agent & Repository	Pipe & Filter	Process Control	Iterative Refinement	Event Based	Layered Systems	Map Reduce	MVC
centralization					X			
distribution	X				X		X	
consistency	X							
parallelism	X							
data dependencies	X							
load balancing	X	X			X			
throughput	X	X	X					
ordering constraints		X						
reuse		X				X		
replaceable		X						
data flow		X						
computation granularity		X		X			X	
communication overhead		X			X		X	
management overhead		X		X				
performance			X			X		X
accuracy			X					X
sample rate			X					
code complexity			X	X	X	X		X
determinism			X					
synchronization overhead				X				
resource utilization				X	X			
scalability							X	X
fault tolerance								X
customizability								X
portability								X

Table 3.6: Forces impacting the PPL algorithm patterns.

Forces	Task Parallelism	Discrete Event	Recursive Splitting	Pipeline	Geometric Decomposition	Speculation
computation granularity	X					X
communication overhead	X					
load balancing	X				X	
synchronization overhead	X					
redundant computation	X					
task dependencies	X	X				
code complexity		X	X			
task uniformity			X			
parallelism		X	X			X
ue utilization			X			
throughput				X		
latency				X		
distribution				X		
generality				X		
customizability				X		
locality					X	
data availability					X	
wrong speculation						X
polling						X
interrupts						X

but with very little overlap between patterns. In fact only 5 *forces* apply to more than one pattern whereas the other 17 *forces* appear in the context of only one pattern.

3.4.2.2 Sorting Tradeoffs

In looking closely at Table 3.7, we see that the same *forces* of: *load balancing*, *memory usage*, *communication overhead* and *exploiting initial distribution* impacting all three sorting patterns. The differences between the three lie in the ways in which they manage these *forces*.

Table 3.7: Tradeoffs in parallel sorting patterns.

	Sample Sort	Quick Sort	Radix Sort
Task	communication overhead vs load balancing	load balancing vs vs communication overhead	load balancing vs memory usage
	exploits initial distribution	exploits initial distribution	load balancing vs exploits initial distribution
Data Control	memory usage	memory usage	communication overhead

For example, *Sample Sort* improved *communication overhead* but at the cost of *load balancing* whereas *Quick Sort* and *Radix Sort* improve *load balancing* with *Quick Sort* giving up *communication overhead* and *Radix Sort* giving up *exploitation of initial distribution* and *memory usage*. In this group of patterns some of the *forces* are addressed with no listed opposing *force*. Specifically, *Sample Sort* and *Quick Sort* address *exploitation of initial distribution* and *memory usage*; *Radix Sort* addresses *communication overhead* with no explicit tradeoff.

3.4.3 Results

In line with the pervasive pattern case study, the goal was to populate the proposed pattern structure in Figure 3.1 and the *force* decision matrix illustrated in Figure 3.2. In this case study we investigated the feasibility of a systematic, *force-centric* perspective of sorting and parallel pattern languages. In this initial analysis, we found the Berkeley layered pattern language to be a sufficient representation of the categorization of the parallel patterns being investigated and so defer to this structure in place of the population of Figure 3.1. Additionally, with this initial analysis we were not able to fully extract the ways in which the tradeoffs play out as opposing *forces* to the same degree we could in the pervasive case study (Section 3.3).

In our analysis of the PPL patterns, we encountered many of the forces section written up by different authors with varying formats and focuses. Some patterns provided an explicit *force-versus-force* list with a supporting discussion for each entry, whereas other patterns provided a focused discussion of each force in isolation. This characteristic limited our ability to translate Tables 3.5 and 3.6 to the RIPL structure of Figures 3.1 and 3.2.

In the analysis of the sorting patterns, the tradeoffs were somewhat easier to extrapolate than the general parallel patterns as demonstrated in Table 3.7. What is not clear from the data in this form is the varying level at which the *forces* are addressed by each pattern. For example, though *Sample Sort* and *Radix Sort* both improve *communication overhead*, *Sample Sort* reduces *communication overhead* relative to *Radix Sort*.

Looking across Tables 3.5 and 3.6 at the *forces* in both the *Structural* and *Parallel Algorithm Strategy Patterns*, nine of the *forces* (bold face font in each table) are mentioned in both layers of the pattern language. We believe that in this form, the information would be valuable to pattern writers in clarifying the following questions:

1. Are compositions of patterns related in terms of tradeoffs?
2. Is this the complete set of forces for a group of patterns?
3. Is a given pattern not considering a *force* that it should be?

The sorting pattern results leads us to question if the binary characteristic of RIPL's systematic approach could be augmented with a deeper decision structure. A decision structure could guide the pattern selection process through choices between opposing *forces* and the subsequent *forces*, providing a more methodological approach to pattern selection.

Figure 3.5 demonstrates one way to provide more quantitative information through RIPL by augmenting it with a relative scale for each force within a pattern. Here, the *force-versus-force* structure provided by RIPL is augmented with '+' and '–' characters to indicate the amount a pattern addresses each *force*. The measurement for each pattern *force* is relative to that of the other patterns listed and is based on the information provided in the pattern language write up. For example, from the first row of this table one can gather that *Quick Sort* has the best *load balancing*, but introduces *communication overhead* in exchange for this benefit. If *communication overhead* is a greater concern, looking at row three we can see that *Sample Sort*

Core Issue	*Dominant Forces*	*Relative Measure & Conflicting Forces*		
		Sample Sort	*Quick Sort*	*Radix Sort*
Non-Functional	load balancing (LB)	+	++ CO	+ MU
Task	memory usage (MU)	++	+	+ EID
Data	communication overhead (CO)	++ LB	-	+ MU
	exploits initial distribution (EID)	++	++ CO, MU	-

Figure 3.5: RIPL applied to Sorting Patterns

addresses this issue the best, but gives up some *load balancing*. Likewise, if *memory usage* is of utmost importance, by looking at columns two and three we can see that *Radix Sort* would be a poor choice of patterns; *Quick Sort* would have to be carefully considered as well, taking into account the other goals of the system and the remaining *forces*.

A similar consolidated approach is used to support pattern composition in the *Three Layer Cake* (Robison and Johnson, 2010). This pattern language provides an overview of patterns representing three styles of parallel programming for shared-memory hardware. The pattern itself is decomposed in terms of *software forces* and *hardware forces* and within each section the *forces* for the three shared-memory, programming styles are discussed. Similar to the sorting pattern language described above, a table provides an overview of the quantitative impact of the *forces* within each pattern in the language. Finally, the *solution* section of this pattern gives insights on composing these programming styles based on the force analysis in the preceding sections.

3.5 Summary

In order to address objective O3:

- **O3.** Discover practical approaches for analysis and comparison of key artefacts.

this chapter provided an analysis of the current development and use of parallel patterns in order to identify challenges and processes associated with pattern use. It is this analysis that identified systematic approaches to pattern analysis for the

purposes of selection. This approach provided the foundation for the RIPL framework that was evaluated as a systematic methodology for parallel pattern analysis in the context of groups of related patterns.

The contribution of this chapter is a systematic, methodological process for the comparison and analysis of patterns within a pattern language based on pattern tradeoffs. The evaluation of this systematic process demonstrated its merits within two related domains, in terms of the pattern selection process along with insights into how patterns could be written to better support their comparison.

An initial analysis of existing pervasive and parallel patterns exposed an overlap in constraints listed across the *forces* sections of different patterns. That is, different patterns address, or are impacted by some similar design and implementation constraints. The common pattern construction, with special attention to the *forces* section, provided the foundation for our systematic, structured approach to pattern comparison.

Further investigation of current work in pattern selection and composition identified the *forces* sections as a key characteristic for comparison. RIPL grew out of this foundation of comparison by *forces*, supporting a systematic process for pattern comparison for selection and composition. The pervasive case study illustrated RIPL's modularized view of pattern tradeoffs within a family and demonstrated the concept of pattern comparison from this structure with support for pattern selection and composition in mind. The pattern case study demonstrated the benefit of this approach in the parallel domain, but requires further work by pattern writers to make this approach feasible.

The application of this structure to a set of pervasive and parallel patterns served to evaluate our systematic approach. A case study involving two families of pervasive patterns allowed us to consider each pattern individually, fully analyse the *forces* section in the context of the whole pattern and then compare across the patterns. This analysis demonstrated both the significance of the *forces* section of the patterns and the ability to break them down into a structured set of tradeoffs on which to base a meaningful comparison. The case study applying our proposed approach within the parallel pattern domain demonstrated the need for a uniformity in the way patterns are written up in order to support such a structured approach. With parallel pattern languages being in their inception and multiple researchers authoring the different patterns, uniformity has not yet been achieved or enforced.

This chapter addressed the following items in support of the contribution to this thesis:

- Determined how things have changed since GoF with respect to the development of patterns for parallelism
- Determined what is easy/hard about using patterns for parallelism
- Documented a process through which users select a parallel pattern
- Discovered opportunities to support pattern selection
- Used the opportunities to identify systematic approaches to pattern selection
- Created a framework based on the approaches
- Evaluated in the context of groups of related pattern

The following chapter proposes a conceptual model representative of solutions to parallel problems that intended to apply to key parallel software artefacts in a variety of forms.

Chapter 4

OPA: An Ontology to describe Parallel Applications

The potential to exploit parallelism requires us to rethink how we break down a problem and build up a solution. This chapter proposes the use of an ontology as a mental model for a developer to view relevant parallel software development artefacts that align with the critical elements of a parallel solution.

Targeting research objective **O4** of the thesis, in this chapter we leverage real-world parallel scenarios and existing literature to discover a conceptual model representative of solutions to parallel problems. In support of this objective, this chapter specifically addresses the following research question:

> *Can we derive a useful conceptual model for solutions to parallel problems in the form of an ontology?*

4.1 Introduction

The complexities of parallel programming have long been recognized, but these complexities are now compounded by the growing number of parallel support mechanisms available and the considerable variation between them. In particular, high-level concern identification—and the subtle and sometimes implicit rules of coordination between concerns—must be traceable to implementation to support programmer productivity. Synchronization alone poses difficulties in the management of shared memory between tasks and the navigation of nested and composed occurrences of synchronization mechanisms. Even the so called *embarrassingly parallel* problems still

require some level of coordination. Ultimately, developers must be able to navigate and understand the core concerns and crucial relationships in parallel applications in a language-agnostic way.

In an ontology, concepts, referred to as *classes* or *entities*, along with associated properties and restrictions, offer a formalized way to categorize or classify a set of domain data. The mapping of a given domain ontology to a specific data set creates a navigable knowledge base. In this form, a common point of understanding is created and supports the sharing and reuse of domain knowledge by separating abstraction of the domain from a specific instance representation. Further, the relationships between entities, in conjunction with the associated properties and restrictions, make domain rules explicit and provide a means of analysing these relationships and rules.

The arrival of a new era of programming, where developers must consider the subtleties of parallelism required for modern many-core architectures, calls for a revamping of fundamental conceptual models that support software development processes. As Nygaard and Dahl demonstrated, the creation of a conceptual model for programming should be closely tied to real-world perspectives. They demonstrated that a model developed in this way can link to existing human mental models and capitalize on an understanding of the associated issues in real-world situations to support reasoning about programming complexities. Nygaard further demonstrated the ability to leverage this close tie to real-world scenarios for pedagogical purposes (Nygaard, 2002).

4.2 Proposal

We propose the use of a common ontology called *OPA* (*an Ontology to describe Parallel Applications*), that maps to parallel problems and solutions both at a high level of abstraction to leverage existing mental models in understanding complexities, and at the level of lines of code for program comprehension tools. This approach is intended to demonstrate the subtle complexities of concurrency in real-world scenarios, creating a conceptual model of parallelism that leverages the reasoning capabilities of an ontology and translates to the comprehension of parallel software.

The entities of the ontology are drawn from Berkeley literature and experiments with educational-based activities revolving around real-life parallelism (Gunion, 2009). Results from these case studies form the top level of our proposed ontology: *computation* and *communication*. Significant overlap is found in this coarse grained entity

classification, calling for a refining and deepening of the ontology. An analysis of existing literature, coupled with educational case study results, provides a basis for this refinement.

In this unified form, the mapping of the ontology to an abstract scenario or software implementation supports both comprehension as well as the ability to compare the implications of parallelism in different solutions. This abstract description is intended to be general enough to apply to other representations of a parallel solution including design patterns and source.

The following subsections provide the details of a two-part analysis of three activities presented to a group of seventh grade students (Section 4.2.1) and a deepening of the ontology to ensure an alignment with a developer's code perspective (Section 4.2.2).

4.2.1 Identifying Coarse-Grained Entities

In order to identify an initial set of entities that represent a parallel problem and solution, we draw from Berkeley's seminal survey of the domain (Asanovic *et al.*, 2006) leveraging *Dwarfs* as a way to design and evaluate parallel programming models. Similar to the thesis of this work, Berkeley is concerned with the clarity of expressing parallel computation and identifies the need for a higher level of abstraction for reasoning about parallel application requirements. The Berkeley approach defines a number of *Dwarfs*, which each capture a pattern of *computation* and *communication* common to a class of applications. While the Berkeley parallel program classification is based on dynamic program behavior, they do leverage this result to further compare programming models from a static perspective and therefore we align the coarse-grained entities of our ontology with this decomposition into core units of *computation* and *communication*.

The following subsections provide an evaluation of the suitability of these coarse-grained entities, grounded within a case study on using three activities to introduce key concepts of concurrency to grade school students (Gunion, 2009). The activities used real-world and storyline style scenarios including: (1) two people washing a set of dishes, (2) two people and two movie ticket queues and (3) an altered version of the classic pedagogical, concurrency scenario of the dining philosophers (Roscoe *et al.*, 1997). Each scenario allowed for a range of concurrency to be introduced to the possible solutions. Different strategies for solving the problem introduce different

levels of complexity in terms of the execution of a solution, but the core activity remained the same. We refer to the general notion of *tasks* associated with each activity as *computation*, and multiple students participating in each activity required some level of *communication* between individuals.

The following three subsections specifically investigate the ways in which *computation* and *communication* apply to the solutions defined by the participating students for each of these three scenarios, and the associated consequences encountered when concurrency is introduced.

4.2.1.1 Dishwashing Scenario

The *Dishwashing* scenario is described as a stack of dirty dishes that need to be cleaned, dried and put away by two people. In this analysis, we take the following three solutions proposed by students for the *Dishwashing* scenario and break them down in terms of the coarse-grained entities of *computation* and *communication*.

- **Solution A:** One person washes the dishes and the other person dries and puts away the dishes.
- **Solution B:** One person washes the dishes, the other person dries the dishes and both people put the dishes away.
- **Solution C:** Both people wash the dishes, both people dry the dishes and both people put the dishes away.

Though the students proposed a variety of solutions for distributing the work, the core *computation* remained the same, with differing levels of communication in each scenario.

Computation: Washing a dish, drying a dish and putting a dish away, for all of the dishes in the stack, was identified by all students as the core elements of *computation* in this scenario. These small distinct pieces of *computation*, or sub-tasks, combine to make up the larger complete task.

Communication: The exact *communication* between participants in this scenario is solution dependent, but in general students identified a visual *communication* protocol between participants as one person handed off a dish to the next person. While these *handoff* points varied, the idea that one individual must complete their portion of the *computation* before the next could take that dish dictated a partial ordering of sub-tasks.

Concept overlap – computation/communication: Though the students seemed to immediately recognize elements of *computation* versus *communication*, they are tightly coupled. In fact, the completion of one part of the *computation* is a form of *communication* to the student's partner in this work.

4.2.1.2 Movie Ticket Scenario

The *Movie Ticket* scenario was described as the problem of two friends wishing to purchase tickets for a popular movie at a theatre with two long ticket queues. This was a slightly fabricated problem, in which communication was limited by a lack of visual contact between queues. Students had to come up with solutions that involved sticking together or splitting up across the queues, and the consequences of each option. Again, we take the following three solutions considered by students for the *Movie Ticket* scenario and break them down in terms of *computation* and *communication*.

- **Solution A**: Each person goes to a different ticket queue and the first person to the front purchases tickets for both of them.
- **Solution B**: Both people wait in the same ticket queue together and purchase individual tickets.
- **Solution C**: Each person goes to a different ticket queue and each purchases their own individual ticket.

Computation: The *computation* in this scenario is simply the act of purchasing a ticket. Depending on the solution, multiple tickets can be bought by one person or a single ticket by each person.

Communication: In cases where students decide the quickest way to get tickets is to split up, they soon realize the associated consequence of possibly buying too many tickets. This race condition, coupled with the non-deterministic processing of the respective queues, required students to consider creative ways to communicate beyond the subtle visual cues leveraged in the *Dishwashing* scenario.

Concept overlap – computation/communication: The overlap between *computation* and *communication* in this scenario is solution dependent. In the case of students making use of the two ticket queues, *computation* can be performed across the two ticket booths but this introduces the need for *communication*. In the case of

the students staying together in one queue, *communication* is not necessary, but the concurrency is sacrificed.

4.2.1.3 Dining Philosophers Scenario

This scenario was renamed *Knights & Forks* and altered from its original context of philosophers, instead described as five knights sitting at a round-table with a single fork between each pair of side-by-side knights. A knight requires acquisition of two forks in order to eat, and when not eating he is thinking independently. Students were asked to come up with solutions to manage access to the forks shared between the knights. Again, we take the following three solutions posed by students for the *Dining Philosophers* scenario and break them down in terms of *computation* and *communication*.

- **Solution A**: Taking turns around the table, two knights eat at a time, requiring at least three rotations for every knight to eat.
- **Solution B**: Ask one knight to leave and the two pairs of knights remaining alternate turns eating.
- **Solution C**: Find five more forks and each knight can eat whenever they want.

Computation: The *computation* in this scenario can be identified as the actions of eating and thinking. Though attempting to acquire a fork is arguably a sub-task, it was not considered to be first-class *computation*.

Communication: In this case, the *communication* is aided by the visual cue of a fork being available for use by a knight wishing to transition from thinking to eating. *Communication* is required between the participants attempting to acquire a fork and his two adjacent participants.

Concept overlap – computation/communication: As with the dishwashing scenario, visual cues (such as an adjacent knight releasing or acquiring a resource) dictated an ordering such that the end of one sub-task could trigger the beginning of another.

In this activity the consequences of concurrency that are particularly problematic came to the forefront. Students encountered issues of race conditions and deadlock through role playing, and came up with multiple strategies for managing the shared

resource to facilitate each participants' chance to eat. Each of these strategies required some form of *communication* between participants based on the pattern of *computation*, that is, what order they will eat and think.

4.2.1.4 Summary

These pedagogical activities served to help us identify common ways in which these students thought about the problems associated with concurrency. In each case, students immediately identified tasks/sub-tasks, and coupled these with the often implicit communication mechanisms, including visual cues. Table 4.1 highlights this breakdown for each activity.

Table 4.1: Linking real-world examples to *computation* and *communication* entities

Entity	Scenario: Dishwashing (DW)	Scenario: Movie Theatre (MT)	Scenario: Knights & Forks (KF)
Computation	wash, dry, put away	purchase a ticket	eat, think
Communication	visual, event driven	absent, problematic	visual, mutual exclusion

4.2.2 Refining the Ontology

The preliminary discovery of entities described in Section 4.2.1 calls for a more fine-grained classification, while still retaining the high-level conceptual model of *computation* and *communication*, as they are a useful foundation.

For a more precise representation of a parallel solution, we believe a finer-grained abstraction is best achieved with the hierarchical structure provided by an ontology. Further, we hope to derive an ontology that would apply interchangeably to both activities, code and other representations of parallel solutions such as patterns.

Berkeley's original white paper on parallel computing research (Asanovic *et al.*, 2006), grounded in embedded applications research (Plishker *et al.*, 2004) and high-performance computing (Pancake and Bergmark, 1990), establishes a set of five *critical parallel tasks* to form a foundation for language comparison. Support for the following set of tasks: *task identification*, *task mapping*, *data distribution*, *communication mapping* and *synchronization* was subsequently compared across ten programming models. The results showed that the support ranged from requiring the

programmer to make explicit decisions for all tasks in order to achieve efficiency, to models that handle all decisions for the programmer for the purposes of productivity.

Combining these *critical tasks* leveraged in the Berkeley work with our observations from real-world parallel problems we propose a more fine-grained breakdown of the *computation* and *communication* entities, extending the conceptual model proposed in Section 4.2.1. The following two subsections identify this finer-grained set of entities and how they relate to *computation* and *communication* in order to establish a multi-tier ontology hierarchy.

4.2.2.1 Computation

Considering the pure *computation* within a solution to a parallelizeable problem, in our observations of educational activities we identify two finer-grained key entities: *task* and *sequential*. *Task* encompasses the actual *computation* to be performed in parallel, or its direct invocation. The *sequential* entity is the portion of the computation that cannot be parallelized, often including code to setup the parallel computation across compute units or the gathering of results from compute units. When this setup or gathering portion of the computation takes more time than the actual task, it can be considered the limiting factor in Amdahl's law as described in the introduction of this thesis.

The *task* entity aligns with the parallel programming model originally identified in embedded applications (Shah *et al.*, 2003) and high-performance computing (Pancake and Bergmark, 1990), in which one of the critical steps in parallelism is 'the division of the application into parallel tasks'. This division of task is also witnessed in how the students naturally develop solutions at an abstract level in the *Dishwashing* and *Movie Ticket* activities.

The *sequential* portion of the code in the educational activities varied substantially in terms of the decisions made to maximize parallelism and consider tradeoffs associated with those decisions. For example, the decision of whether or not to split across lines at the movie theatre severely impacts the identification and articulation of the *sequential* portion of the solution. Specifically, a solution involving splitting into two lines would augment the sequential portion with the computation required to set up the split and communicate between participants when tickets.

4.2.2.2 Communication

Communication is necessary between tasks that must work together to compute a result, where the level of *communication* is dependent on how tightly coupled these tasks are in terms of shared data or state. The finer-grained tasks of *data distribution* and *synchronization* from the Berkeley group directly facilitate and regulate *communication* between tasks. Data can be distributed through shared memory or placement of the data within a worker's local memory space, providing direct communication between compute units; synchronization mechanisms tend to regulate *communication* by coordinating access of compute units to shared state. This finer-grained breakdown of communication issues is further supported by Berkeley's identification of *distribution of data to memory elements* and *inter-task synchronization* as characteristics of parallel support mechanisms (Asanovic *et al.*, 2006).

While the activities given to the students were not examples of pure data parallelism, the same issues of *data distribution* and *synchronization* came into play within the problems and solutions. The *Dishwashing* scenario being more of a dataflow style of concurrency used a shared buffer for *communication*. That is when a shared dish was done being washed, it was placed in the buffer for the next task to begin.

The forks in the *Knights & Forks* scenario is also an example of shared state that a task requires to access the shared data (the food). This concept of a shared buffer or shared state introduces *tasks* that require *synchronization* around a shared resource to access a shared, non-distributed set of data. The *Movie Ticket* scenario was a good example of task parallelism in which *data distribution* is across two data sources. The inability for students to introduce an explicit *synchronization* mechanism around this distributed data allowed them to reason about the consequences of accessing shared data without communication such as ending up with the wrong number of tickets.

4.2.2.3 Minimizing the Overlap

Based on the analysis of Sections 4.2.2.1 and 4.2.2.2, we can somewhat logically parcel out the entities of pure *computation* and *communication* in the activities as: *sequential*, *task*, *synchronization* and *data distribution*, but these four entities do not encompass all aspects of a parallel solution. Specifically, *task mapping* and *communication mapping* in Berkeley's *critical tasks* are not accounted for in the above breakdown. In each of the educational activities, students identified ways to coordinate both *tasks* and access to *data*. Drawing from the students' problem-solving strategies, we link

the *task* and *communication mapping* to the *coordination* that emerged as key factors in their solutions. As *communication* already lies at the upper level of our ontology it is not repeated at the lower level and so we end up with just two additional entities of: *task coordination* and *data coordination*.

While *task* and *data distribution* are very distinct entities, *task coordination* and *data coordination* are more tightly coupled. They both involve provisioning of resources:

- *task coordination* handles resource provisioning primarily for *computation*, but also requires *communication* to tasks
- *data coordination* handles resource provisioning primarily for *communication*, associated with *computation*

Computation and *communication* each play a part in both *task* and *data coordination* and so an overlap emerges in the ontology, as illustrated in Figure 4.1. The solutions to real-world scenarios support this outcome by demonstrating very little conscious coordination by the students, but as discussed above the overall solutions they proposed contained combinations of *computation-* and *communication-based* elements that introduced implicit coordination. It was observed in (Gunion, 2009) that *coordination* emerges as an agreed upon protocol in the suggested solutions within the activities. These protocols deal with coordinating access to shared resources (*data coordination*) or coordinating the execution of separate tasks (*task coordination*).

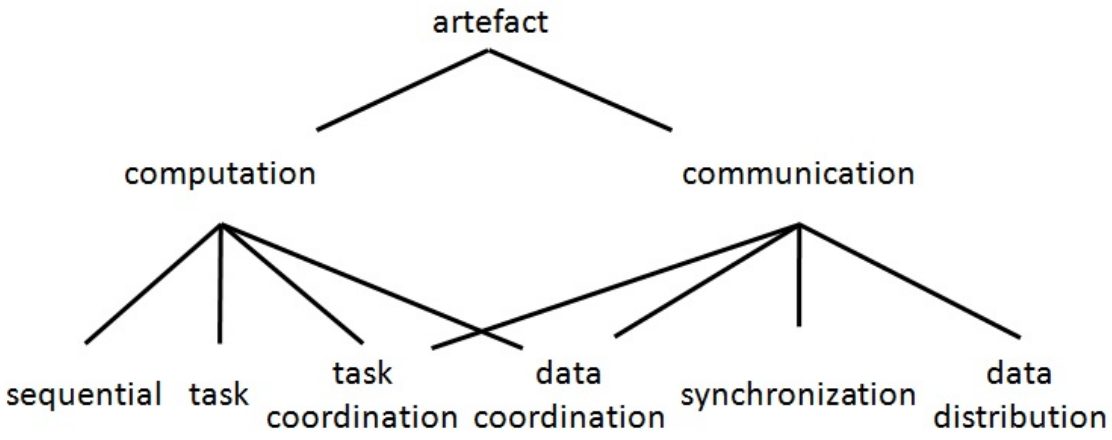

Figure 4.1: Relationships between *computation* and *communication* entities.

As pointed out above, *task* and *data coordination* both center around resource provisioning and management based on the work to be done. This relationship between these two entities can be represented within the ontology by extending the hierarchy to include *coordination* as a top-level entity along with *computation* and *communication*.

With the identification of *coordination* as a third entity, we must consider its relationship to the other fine-grained entities that make up the ontology. While *sequential*, *task* and *data distribution* entities have no direct relationship to *coordination*, *synchronization* is key to the *coordination* of access to system resources: it can be explicit and tightly coupled within an application, or concealed and handled within the background of a support mechanism. *Synchronization* manages resource access by communicating between system components, establishing a strong connection to both *communication* and *coordination* where *communication* is in service of the resource management and provisioning.

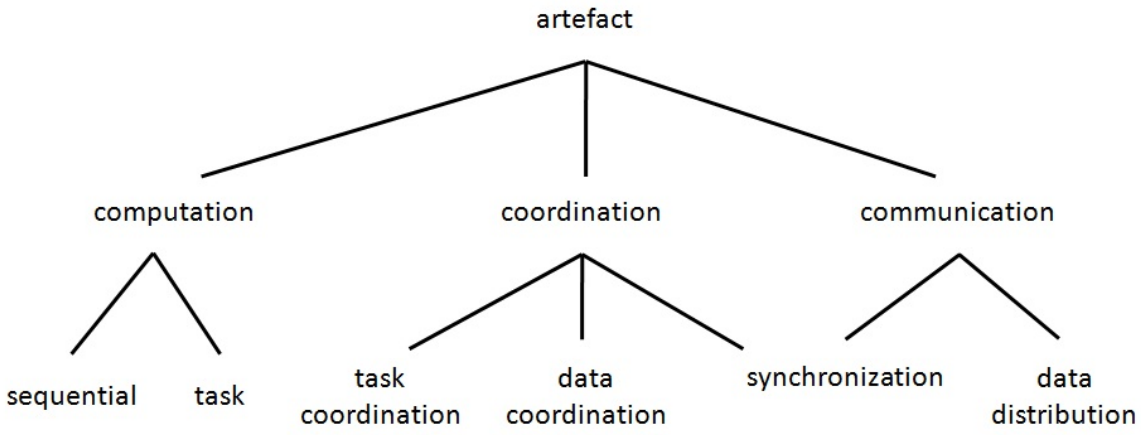

Figure 4.2: The emergence of *coordination* as a third high-level entity.

Figure 4.2 shows this extended ontology based on the relationship of *coordination* to existing low-level entities with the overlap of synchronization represented through multiple inheritance. This varying granularity of perspectives supports not only the comprehension of fine-grained entities in isolation but also a holistic view of entity relationships. For example, in order to understand the three *coordination* entities outlined in this work within an application we must understand the relationships that are only visible when all entities are considered together.

We believe these perspectives can work in synergy to support the linking of low-level implementation details to high-level abstractions and patterns. Table 4.2 provides a summary of how each of these six entities would map to a language independent, software implementation. *Sequential* would be algorithm dependent, but in our experience, it tends to be associated with the setup phase. *Task* maps to the *computation*, whereas *task coordination* corresponds to resource and context management. *Data coordination* maps to memory allocation, partition sizes and buffer creation, whereas *data distribution* would be the actual copying or assignment of data. Finally, *synchronization* would map to the application of any provided or derived synchronization primitives.

Table 4.2: Mapping fine-grained entities of OPA to implementation

Entity	Implementation Description
sequential (SEQ)	computation coordination of computation
task (TSK)	direct computation computation invocation
task coordination (TC)	resource allocation wrapper functions arguments and context queue management
data coordination (DC)	memory allocation intermediate buffer creation partition function invocation partition size management
synchronization (SYN)	synchronization primitives barriers
data distribution (DD)	data copying partition function data assignment

4.3 Evaluation

The evaluation of OPA is based on a full mapping of the full ontology to the real-world scenarios described in Section 4.2.1 (Section 4.3.1) and an assessment of the method used to develop OPA (Section 4.3.2).

4.3.1 Full Ontology Mapping

In this section we reflect back on the educational activities overviewed in Section 4.2 to consider the applicability of the finer-grained mapping proposed in Section 4.2.2. Here we outline the results of this fine-grained mapping to the students' solutions to concurrency problems within the pedagogical activities.

Table 4.3 provides a mapping of OPA's finer-grained entities proposed in Section 4.2.2 onto the problems posed in the educational activities and the students' solutions. The mapping in this table assigns *sequential*, *task* and *synchronization* entities to aspects of the problem description whereas *task coordination*, *data coordination* and *data distribution* entities are solution-specific.

Table 4.3: Full mapping of OPA ontology onto activity solutions

<table>
<tr><th>Entity</th><th>Scenario:
Movie Theatre (MT)</th><th>Scenario:
Dishwashing (DW)</th><th>Scenario:
Knights & Forks (KF)</th></tr>
<tr><td>SEQ</td><td>wait in line and
purchase a ticket</td><td>wash, dry, put away
one dish at a time</td><td>each knight takes
a turn eating</td></tr>
<tr><td>TSK</td><td>purchase a ticket</td><td>wash, dry, put away</td><td>eat, think</td></tr>
<tr><td>TC</td><td rowspan="3">see Table 4.4</td><td rowspan="4">see Table 4.5</td><td rowspan="4">see Table 4.6</td></tr>
<tr><td>DC</td></tr>
<tr><td>DD</td></tr>
<tr><td>SYN</td><td>not synchronized</td></tr>
</table>

In the case of all three scenarios, the *task* entity (TSK Row) maps to the details associated with the high-level *computation* entity described in Table 4.1 in Section 4.2.1.4. The *sequential* portion (SEQ Row) for each scenario is populated with a serialized version of the *task*.

The varying levels of *synchronization* (SYN Row) used in each scenario is highlighted, starting with as little as no synchronization in the *Movie Ticket* scenario solutions. The *Dishwashing* scenario synchronizes placement and removal of a dish from a shared buffer while the *Knights & Forks* scenario synchronizes the access to a fork shared between two participants.

Tables 4.4, 4.5 and 4.6 provide an overview of the *task coordination*, *data coordination*, *synchronization* and *data distribution* involved in the three different solutions proposed by seventh grade students for the three scenarios (MT, DW, KF). In this form, Tables 4.4, 4.5 and 4.6 isolate the tradeoffs between the solutions and support

the comparison of the consequences of design decisions.

In the first solution to the *Movie Ticket* scenario (MT 1) both people are trying to buy two tickets in separate line ups. In the second and third solutions (MT 2 and MT 3) both people are trying to buy their own individual tickets, where in the second solution the two people stick together in the same queue and in the third solution they split up across the two queues. The issues with lack of *synchronization* become apparent when students realize that, while the first solution may potentially be the fastest, they could end up with four tickets to the movie. A more complex solution that leverages *communication* between parties could facilitate *synchronization* to eliminate the chance of too many tickets being purchased.

Tables 4.5 and 4.6 provide a similar result to that of the *Movie Ticket* mapping described above, but also demonstrates the variation in synchronization. In both cases, access to a shared resource leverages visual communication for *synchronization*.

In the *Dishwashing* scenario, the need for *synchronization* is centered around ensuring a buffer or bin is not empty when looking to remove a dish, or too full when looking to add a dish. In the *Knights & Forks* mapping, *synchronization* is necessary for any fork that is shared by more than one knight. These two examples illustrate, at a pedagogical level, the difference between various *synchronization* mechanisms.

Table 4.4: Solution-specific ontology mappings for *Movie Ticket* scenario

	MT 1	MT 2	MT 3
TC	each person assigned a queue, first to the front purchases two tickets	both people go to same queue and purchase tickets individually	each person assigned to a queue to purchase tickets individually
DC	ticket access at two booths	ticket access at one booth	ticket access at two booths
DD	tickets at each booth	tickets at one booth	a ticket at each booth

4.3.2 Validation of Ontology Development Process

Noy and McGuinness provide a detailed description of the basic steps of ontology development based on key structural components that make up an ontology, formal ways to describe those components and the relationships between them (Noy and McGuiness, 2001). We use this process as a point of reference to validate the development of OPA.

Table 4.5: Solution-specific ontology mappings for *Dishwashing* scenario

	DW 1	DW 2	DW 3
TC	assign washing to one, assign drying to the other, assign putting away to both	assign washing to one, assign drying & putting away to the other	assign drying to both, assign washing to both, assign putting away to both
DC	allocate 'washed dish bin' to hold washed dishes allocate 'clean dish bin' to hold clean dishes	allocate 'washed dish bin' to hold washed dishes	allocate 'washed dish bin' to hold washed dishes allocate 'clean dish bin' to hold clean dishes allocate 'dry dish bin' to hold dry dishes
SYN	synchronization on: 'washed dish bin' and 'clean dish bin'	synchronization on: 'washed dish bin'	synchronization on: 'washed dish bin', 'clean dish bin' and 'dry dish bin'
DD	across: 'washed dish bin' and 'clean dish bin'	across: 'washed dish bin'	across: 'washed dish bin', 'clean dish bin' and 'dry dish bin'

Table 4.6: Solution specific ontology mappings for *Knights & Forks* scenario.

	KF 1	KF 2	KF 3
TC	coordinate a schedule of: two people eating and three people thinking	choose who to get rid of and coordinate a schedule of: two people eating and two people thinking	everyone eats when they want
DC	a fork between two knights	a fork between two knights, one pair has an extra fork	each knight has two forks each knight has two forks
SYN	synchronization on: each fork	synchronization on: each fork	no synchronization
DD	five forks for five knights, place one fork between each pair of knights	five forks for four knights, place one between each pair of knights and give the extra fork to one pair of knights	ten forks for five knights, place two forks for each knight

Recall that a *class* or *concept* is the basic building block of an ontology and these together with the relationships between them form the backbone of an ontology.

To provide detailed information within this type of knowledge base, each class has both *slots*, which are the roles or properties of the class and *facets*, which define the restrictions on a class.

A general approach for developing a new ontology is described by Noy and McGuinness as: "defining classes in the ontology, arranging the classes in a taxonomic (subclass/superclass) hierarchy, defining slots and describing allowed values for these slots, filling in the values for slots for instances".

The authors go on to describe this process more formally in a *simple knowledge-engineering methodology* comprised of seven steps. In an attempt to evaluate our approach to developing OPA, for each of these seven steps we provide: 1) a definition of the process with applicable terminology as provided by Noy and McGuinness and 2) how our approach does and does not align with this process.

Step 1 of the proposed methodology is to *determine the domain and scope of the ontology*. This step deals with establishing: a) the set of knowledge that the ontology will apply to, b) what the ontology will be used for, c) what types of questions the knowledge base will address and d) who will use and maintain it. OPA was developed to provide a conceptual model for people programming for multi-core architectures and so the parallel programming domain was identified to be the scope of our ontology. The intent of the initial development of OPA was to provide a conceptual model that would apply to all types of parallel software artefacts identified in the *Rupture* model. The goal of OPA is to provide support for the comparison of differences across different types of software artefacts. In terms of use, OPA is intended to be used by both developers and educators. As OPA was developed as a proof of concept, the actual mapping of the ontology onto artefact instances and maintenance was left as a manual process performed by the user.

Step 2 says to *consider reusing existing ontologies* in order to allow for future reuse and to integrate with other related ontologies in terms of vocabulary. While we see this as a viable step, in an extensive search of existing work in the parallel and ontology domains we could not locate any developed or partially developed ontologies relevant to the parallel domain.

Step 3 tells ontology developers to *enumerate important terms in the ontology* in a brainstorming method. We took a much more research-based approach in the development of OPA, calling on related work from experts in the parallel software domain to identify key terms.

Step 4 advises the developer to *define the classes and the class hierarchy* in one of

three methods: 1) *top-down*, 2) *bottom-up* or 3) a *combination* of the first two. The *top-down* approach starts at the most general classes and then further specializes, whereas the *bottom-up* approach starts with the most specific classes and combines them into more generalized groups to develop the upper layer classes. The *combination* approach identifies both top-level and bottom-level classes at the same time and then relates the two through a middle level of classes. The *combination* approach describes the development of OPA. Specifically, we identified the high-level classes of *computation* and *communication* first, followed by the low-level classes while considering the relationship of the low to the high-level entities at the time of discovery.

Step 5 calls for the developer to *define the properties of classes—slots*, where the *slots* describe the internal structure of the class. These *slots* are intended to be drawn from the terms generated in the brainstorming session (*Step 3*) that were not identified as classes (*Step 4*). Our process to develop OPA did not follow this format strictly. The complexity of the domain, combined with the wide scope encompassed by the OPA classes, forced a more descriptive and parallel software artefact definition of OPA slots. The software *implementation descriptions* for each class outlined in Table 4.2 define the properties of each of the low-level OPA classes in terms of their application to source artefacts specifically. This table demonstrates a much more detailed description of each of the classes in OPA than the suggested *terms* identified in *Step 3*, and highlights the need for different *slots* for the different types of software artefacts.

Step 6 calls for the developer to *define the facets of the slots* where the *facets* establish rules associated with the types of instances to which a specific class could be mapped. We have not yet defined *facets* for OPA, as this proof-of-concept is intended to map to and support reasoning about changes to already developed software artefacts. That said, the identification of *facets* for OPA that would apply to the different software artefacts could help with error and violation detection.

Step 7 calls for the *creation of instances* of the ontology senilities. In this development phase of OPA we map instances of the solutions to pedagogical activities to the ontology. Chapter 5 continues with this step, evaluating OPA in the context of other software artefacts.

The development of OPA was an iterative process in which the term identification (*Step 3*) was interleaved with the definition of classes(*Step 4*) and slots(*Step 5*), with the results of the class and slot definition feeding back into the identification of new terms and subsequent classes. This method aligns with Noy and McGuinnesses'

description of ontology development as "necessarily an iterative process" in their set of fundamental rules for ontology design.

4.4 Summary

In order to address objective O4:

- **O4.** Discover a conceptual model representative of solutions to parallel problems.

this chapter provides an investigation of real-world scenarios developed from a pedagogical standpoint and their relationship to key elements of parallel software as identified by experts in the area. A further analysis of the results of this investigation unearthed the critical tasks of a parallel program and the relationships between them providing a foundation for OPA. An evaluation of OPA's ability to describe parallel applications is performed in terms of its applicability to solutions to real-world scenarios and the ability to reveal tradeoffs between these possible solutions.

The contribution of this chapter is the development of OPA as a new conceptual model, specifically for the parallel domain in the form of an ontology. We began identifying real-world scenarios that demonstrated issues of parallelism in order to ground this model in human conceptual thinking. This approach, influenced by Nygaard and Dalh's work with pedagogy and object-oriented programming, was tightly tied to Berkeley's initial survey of the parallel programming domain.

In terms of evaluation, the key elements of *computation* and *communication* that make up the execution of a parallel program, as drawn from Berkeley's observations, were shown to apply to student solutions to parallel-specific scenarios posed in pedagogical activities.

In considering Berkeley's static perspective of parallel programs and their list of *critical tasks* that must be addressed by a parallel programming model, we identified a more fine-grained set of entities that represent a parallel solution. The relationships between the low-level and high-level entities were established, identifying overlap between entities. This overlap was minimized by introducing an additional high-level entity of *coordination*. This final set of entities and relationships form the basis for OPA as the proposed parallel ontology.

The evaluation of this proposed ontology included its full mapping onto three pedagogical activities demonstrating the ability of the ontology to reveal the tradeoffs

by isolating similarities and differences between various solutions. A further evaluation of the process for developing OPA was guided by a process set out by ontology experts. OPA was vetted and accepted as a viable ontology by the ontology community (Gibbs and Coady, 2010a) and accepted by the parallel pattern community as support for pattern mining (Gibbs *et al.*, 2009). The following chapter provides further evaluation of OPA in terms of its mapping to artefacts at multiple levels of the software life cycle including code and patterns. This chapter accomplishes the following items in support of the contribution to this thesis:

- Determine real-world scenarios that demonstrate issues of parallelism.
- Discover key elements that make up the execution of a parallel program that align with conceptual models.
- Discover critical tasks of a parallel program from existing literature.
- Discover associations between execution elements and program tasks.
- Use the identified key elements, critical tasks and associations to define the entities and relationships that conceptually describe the solution to a parallel problem.
- Create an ontology based on the defined entities and relationships.
- Evaluate in terms of how this ontology maps to the solutions of real-world parallel problems.
- Evaluate the ontology in terms of its ability to reveal the tradeoffs in terms of similarities and differences between solutions.

The following chapter continues to evaluate the conceptual model of OPA in terms of its ability to map to key parallel artefacts in various forms in a consistent manner.

Chapter 5

Case Study: Mapping OPA to Rupture Artefacts

This chapter leverages OPA, the ontology proposed in Chapter 4, to evaluate the proposed use of an ontology in the parallel software development process. Targeting research objective **O5** of the dissertation, this chapter maps the proposed ontology to multiple artefact types and uses these mappings in a method for comparison and analysis both within and across artefact types. The contribution of this chapter has been vetted in the parallel pattern community (Gibbs *et al.*, 2009). In support of this objective, this chapter, will specifically address the following research question:

> *Can an ontology provide a conceptual model that supports analysis for software development tasks on individual artefacts to provide knowledge transfer between artefacts?*

5.1 Introduction

In this case study, the implementation of two parallelizeable problems are leveraged to demonstrate the application of the OPA ontology to two types of Rupture software artefacts: source code and design patterns. Specifically, three implementations of a simple reduction or summation of values, three implementations of the Fast Fourier Transform (FFT) (Cooley and Tukey, 1965) algorithm and the *MapReduce* design

pattern, as it applies to the implementation of the simple reduction, form the basis of this case study.

The first two parts of this case study focus on the mapping of OPA described in Chapter 4 onto the source code of the simple reduction (Section 5.2) and the FFT (Section 5.3) implementations. These experiments provide a two-part analysis: 1) outlining the qualitative results of an analysis of the source in terms of OPA's coarse-grained entities and 2) a quantitative overview of the manual classification of lines of code in terms of OPA's fine-grained entities. The third part of this case study applies OPA to the MapReduce design pattern, the pattern that describes the simple reduction implementations overviewed in the first portion of this case study. This experiment capitalizes on the significance of the *forces* as identified in Chapter 3 and maps the fine-grained entities of OPA to the *forces* of the MapReduce pattern.

5.2 Mapping OPA to Source Artefacts: A Simple Reduction

Arguably a *poster child* for parallelism, here we consider the implementation of what is commonly called a *simple reduction* which involves the summation of a large set of values. The implementations surveyed here employ three different parallelization support mechanisms: a framework for MapReduce (Dean and Ghemawat, 2004), OpenCL (Apple, 2008) and CUDA (NVIDIA Corporation, 2010). The MapReduce framework allows developers to write specialized `map` and `reduce` functions, whereas CUDA and OpenCL localize the code that will execute on the parallel compute units in a separate module referred to as a `kernel`. CUDA targets GPU architectures, whereas OpenCL provides a generalized interface that can take advantage of both CPU and GPU multi-core architectures.

5.2.1 Qualitative Analysis of Simple Reduction Implementations

Though the implementation of the simple reduction algorithm varies considerably between MapReduce (Figures 5.1 and 5.2), CUDA (Figures 5.3 and 5.4) and OpenCL (Figures 5.5 and 5.6), we begin by considering each in general terms of *computation* and *communication*, from the highest layer of OPA. This initial analysis is followed

```
1  void sumarray_map( map_args_t * args ) {
2    int nChunkSize  = args->length;
3    int * miniArray = (int*) args->data;
4    int intermediate_sum = 0;
5    for(int i=0;i<nChunkSize;i++) {
6      intermediate_sum += miniArray[i];
7    }
8    ...
9    emit_intermediate( key, val, sizeof( int* ) );
10 }
11 void sumarray_reduce(void *key_in, void ** vals_in,
12                       int vals_len) {
13   int nElements = vals_len;
14   int ** p_array = (int**) vals_in;
15   ...
16   int sum = 0;
17   for(int i=0;i<nElements;i++) {
18     sum += p_array[i][0];
19     delete p_array[i];
20   }
21   ...
22   int * val = new int;
23   *val = sum;
24   emit( key, val );
25 }
```

Figure 5.1: MapReduce functions for a simple reduction

by the identification and analysis of the overlap between these high-level entities as it plays out within the source code.

Computation: The user defined `map` and `reduce` functions, shown in Figure 5.1 and 5.2, constitute the application-specific code to be parallelized by the underlying framework. These functions house the core *computation* of this algorithm, revealing the two loops that constitute the main calculation (Figure 5.1: lines 5-7, 17-20). The corresponding CUDA and OpenCL kernel modules similarly contain the *computation*, albeit couched in a slightly more optimized and architecture-specific form than the MapReduce implementation.

Figures 5.3 and 5.5 contain roughly corresponding code segments from the OpenCL and CUDA kernel modules, with a similar mix of global data access (Figure 5.3: line 3, Figure 5.5: line 3), and synchronization (Figure 5.3: lines 6 and 11, Figure 5.5: lines 6 and 10). Arguably, the only line of *computation* with localized impact involves the addition of two elements that are one `stride` length apart (Figure 5.3: line 11,

```
1  int main( int argc, char * argv[] ) {
2    scheduler_args_t sched_args;
3    final_data_t result;
4    int * array;
5    int nElements;
6    ...
7    sched_args.task_data = array;
8    ...
9    sched_args.data_size = nElements*sizeof(int);
10
11   sched_args.map = sumarray_map;
12   sched_args.reduce = sumarray_reduce;
13   sched_args.splitter = NULL;
14   ...
15   sched_args.result = &result;
16   sched_args.partition = NULL;
17   ...
18   sched_args.use_one_queue_per_task = 0;
19   sched_args.L1_cache_size = 8192;
20   sched_args.num_map_threads = -1;
21   sched_args.num_reduce_threads = -1;
22   sched_args.num_merge_threads = -1;
23   sched_args.num_procs = 32;
24   ...
25   if( map_reduce_scheduler( &sched_args ) < 0 ) {
26    fatal("Scheduler had an error. Bailing out.\n");
27   }
28   ...
29   keyval_t * p_pair = result.data;
30   ...
31 }
```

Figure 5.2: MapReduce host code for a simple reduction

Figure 5.5: line 10 respectively).

Communication: MapReduce uses an `emit` function to communicate intermediate results from the compute unit or *worker* to the *host*, which handles the distribution of tasks and data (Figure 5.1: lines 9 and 24). The host uses shared memory (Figure 5.2: line 7) for *communication* of data to the workers (Figure 5.1: line 14) and the final result is retrieved by the host through a shared memory space (Figure 5.2: line 29).

CUDA and OpenCL use memory copying and synchronization mechanisms for *communication* at the kernel level (Figure 5.3: lines 6, 12, 14 and Figure 5.5: line 11 and 14 respectively). At the level of the host, CUDA makes use of library syn-

```
1  while (i < n)
2  {
3    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
4    i += gridSize;
5  }
6  __syncthreads();
7  if (blockSize >= 512) {
8    if (tid < 256) {
9      sdata[tid] += sdata[tid + 256];
10   }
11   __syncthreads();
12 }
13 ...
14 if (tid == 0) g_odata[blockIdx.x] = sdata[0];
```

Figure 5.3: CUDA kernel code for a simple reduction

chronization mechanisms (Figure 5.4: lines 35 and 51) and synchronization supported memory copying (Figure 5.4: lines 23-26). Similarly, OpenCL uses library functions with built in buffers for distributing data to workers (Figure 5.6: line 11-12) and reading back the result from workers (Figure 5.6: line 21).

Concept overlap - computation/communication: For many of these lines of code, it is actually surprisingly difficult to align them with just one entity, as they serve dual purposes in terms of both *computation* and *communication*. Something as simple as an assignment to shared memory demonstrates this property. Similarly, the considerable effort invested to handle resource setup in terms of processing units and memory in each implementation serves this dual purpose: it computes provisioning of resources, in a way that also serves as *communication*.

In MapReduce, data partition size, the number of elements and memory allocation issues are dealt with in the worker functions (Figure 5.1: lines 2-4, 14) and at the host (Figure 5.2: lines 2-5). The MapReduce host deals with the majority of the argument setup for the task execution and data distribution. For example, the selection of a `splitter` and `partition` function (Figure 5.2: lines 13 and 16) provides *communication* of intermediate results and distribution of data. Again, much of the parameter tuning occurs in the host (Figure 5.2: lines 18-23), including cache size and number of threads (Figure 5.2: lines 19-22) and lies in this intersection between *computation* and *communication*.

In OpenCL, the library support abstracts the *communication* details behind the program, context and command queue setup (Figure 5.6: lines 5-6, 15-16). Similarly,

buffer creation to house data for *computation* and for distribution of data for host/-worker *communication* (Figure 5.6: lines 12, 15-16) is supported by library functions.

Both the CUDA (Figure 5.3: line 3) and OpenCL (Figure 5.5: line 3) kernels have single lines of code that perform both *computation* in the form of addition and *communication* through storage of the result to global memory. Additionally, CUDA requires a combination of explicit device setup and use of synchronization mechanisms (Figure 5.4: lines 8-14) to support *communication*.

Though synchronization is not an explicit form of *communication*, we argue that a developer must still consider how the interleaved synchronization calls are managing the implicit *communication* (through shared memory) between the host and the workers and even between the workers themselves. Even the explicit identification of the functions without synchronization (Figure 5.4: lines 23-26) requires a developer to think about the nature of the *communication* involved.

Invocation of the *computation* delegated to the workers also overlaps with *communication* for storage of the calculated results on the host (Figure 5.4: lines 28-29). Synchronization mechanisms are also interleaved within the CUDA host code to ensure correct ordering of workflow through the kernel (Figure 5.4: lines 35 and 51).

5.2.1.1 Summary

Though the source code experiments above consider different problems, we leverage OPA to focus on the common elements that launch the parallelization. That is, the breakdown between the host which handles the distribution of tasks and data across the workers, which correspond to the units that perform the computation.

CUDA provides library support mechanisms for memory allocation and thread synchronization to support *communication*, but in terms of *computation*, identifiers are limited to the global tags that mark the kernel module. This kernel module concept also exists in OpenCL and is identified with a kernel tag.

In addition to this tag, OpenCL provides extensive linguistic support for both *computation* and *communication* issues—making many steps actually more explicit. For example, device and task management functions are provided, as well as those associated with memory allocation, buffer creation and buffer `read` and `writes` provided by the OpenCL library.

MapReduce provides library support for partitioning of data and semantic support in the form of naming conventions for scheduling arguments.

```
1  int
2  main(int argc, char** argv)
3  {
4    char *typeChoice;
5    cutGetCmdLineArgumentstr(argc,
6        (const char**) argv, "type", &typeChoice);
7
8    cudaDeviceProp deviceProp;
9    deviceProp.major = 1;
10   deviceProp.minor = 0;
11   int desiredMinorRevision = 0;
12   ...
13   cutilSafeCallNoSync(cudaChooseDevice(..));
14   cutilSafeCallNoSync(cudaGetDeviceProperties(..));
15   ...
16   runTest<float>(argc, argv, datatype);
17   ...
18 }
19 template <class T> void
20 runTest(int argc, char** argv, ReduceType datatype)
21 {
22   ...
23   cutilSafeCallNoSync(cudaMemcpy(d_idata, h_idata,
24     bytes, cudaMemcpyHostToDevice));
25   cutilSafeCallNoSync(cudaMemcpy(d_odata, h_idata,
26     numBlocks*sizeof(T), cudaMemcpyHostToDevice));
27   ...
28   gpu_result = benchmarkReduce<T>(size, numThreads,
29     numBlocks, maxThreads, maxBlocks, ...);
30   ...
31 }
32 template <class T>
33 T benchmarkReduce(..){
34    ...
35    cudaThreadSynchronize();
36
37   if (useSM13)
38     reduce_sm13<T>(n, numThreads, numBlocks,
39         whichKernel, d_idata, d_odata);
40   else
41     reduce_sm10<T>(n, numThreads, numBlocks,
42         whichKernel, d_idata, d_odata);
43   ...
44   cutilSafeCallNoSync(cudaMemcpy(h_odata, d_odata,
45     numBlocks*sizeof(T), cudaMemcpyDeviceToHost));
46
47   for(int i=0; i<numBlocks; i++) {
48     gpu_result += h_odata[i];
49   }
50   ...
51   cudaThreadSynchronize();
52   ...
53 }
```

Figure 5.4: CUDA host code for a simple reduction

```
1  while (i < n)
2  {
3    shared[lid] += input[i] + input[(i+GROUP_SIZE)];
4    i += stride;
5  }
6  barrier(CLK_LOCAL_MEM_FENCE);
7  #if (GROUP_SIZE >= 512)
8    if (lid < 256)
9      shared[lid] += shared[lid + 256];
10   barrier(CLK_LOCAL_MEM_FENCE);
11 #endif
12 ...
13 if (lid == 0)
14   output[gid] = shared[0];
15 }
```

Figure 5.5: OpenCL kernel code for a simple reduction

```
1  int main(int argc, char **argv) {
2    ...
3    err = clGetDeviceIDs(CL_DEVICE_TYPE_GPU, ...);
4
5    context = clCreateContext(0, 1, &device_id, ...);
6    commands = clCreateCommandQueue(context, device_id, ...);
7
8    const char* filename = ``reduce_kernel.cl";
9    char *source = load_program_source(filename);
10
11   input = clCreateBuffer(context, ...);
12   err = clEnqueueWriteBuffer(commands, input, ...);
13
14   partials = clCreateBuffer(context, ...);
15   output = clCreateBuffer(context, ...);
16   ...
17   err |= clSetKernelArg(...);
18   ...
19   err |= clEnqueueNDRangeKernel(commands, ...);
20
21   err = clEnqueueReadBuffer(commands, pass_output, ...);
22   ...
23   return 0;
24 }
```

Figure 5.6: OpenCL host code for a simple reduction

5.2.2 Quantitative Results of Ontology Mapping

This section outlines the quantitative results of a manual line-by-line mapping of OPA to the three reduction implementations in this experiment. This mapping was guided by the development of OPA from Chapter 4, specifically Table 4.2 that outlined a general mapping of the fine-grained entities of OPA to source code.

Figure 5.7 shows the percentage of code from the three implementations classified as *computation* and *communication*. The dual responsibility of a single line of code, identified in the qualitative analysis in the previous subsection, becomes evident when we consider that for each implementation the combination of *computation* and *communication* counts exceeds 100% of the code base. Figure 5.8 isolates the intersection between *computation* and *communication*, highlighting the fact that the intersection of these two entities, the *coordination* code, is substantially larger than each one in isolation across all three examples.

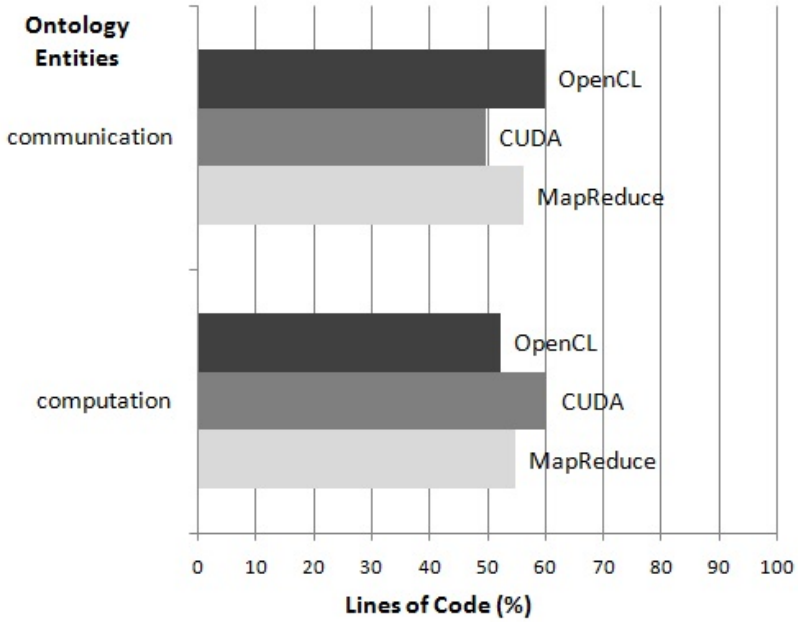

Figure 5.7: *Computation* and *communication* breakdown in three implementations of a simple reduction

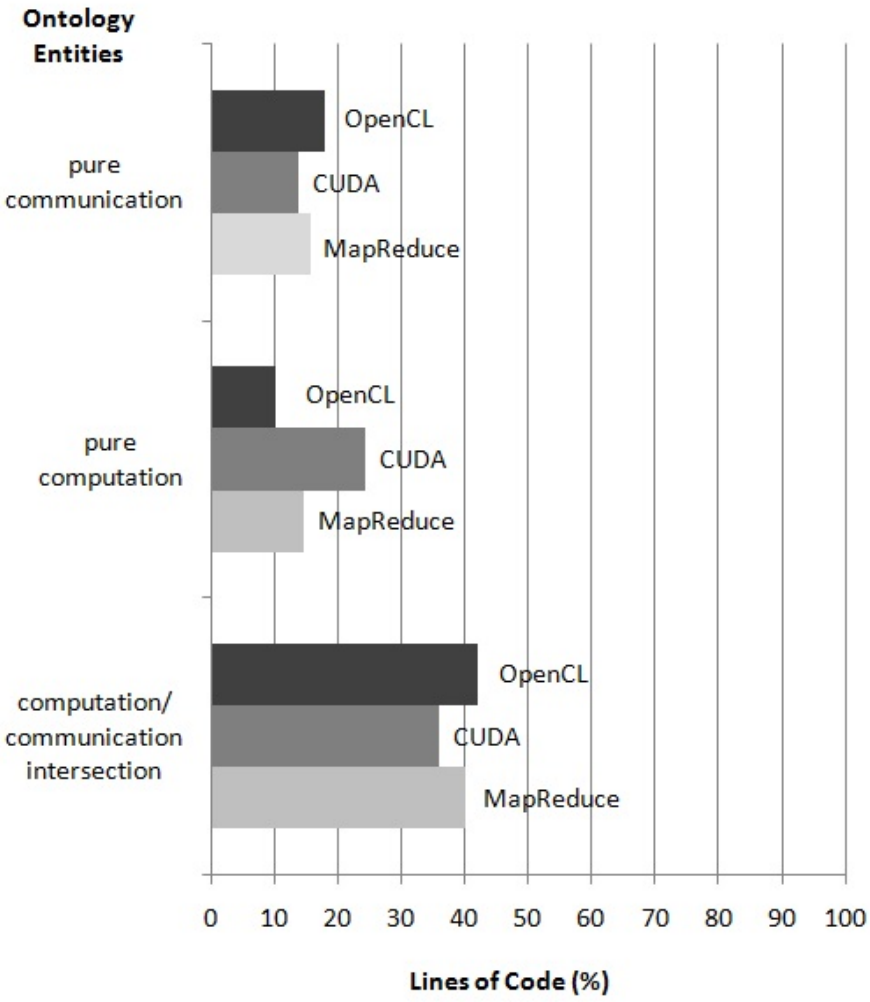

Figure 5.8: *Computation* and *communication* intersection in three implementations of a simple reduction

Figure 5.9 shows the results of mapping OPA's low-level entities to the MapReduce, CUDA and OpenCL implementations of a reduction algorithm. The simplicity of this algorithm is highlighted by the small percentage of code classified as *task*. The CUDA results show a slightly higher percentage of *task* code than that of MapReduce and OpenCL, but this is due to the fact that the CUDA reduction provides six different compute kernels, each providing an alternative optimization strategy.

If we consider these results from the perspective of the top-tier entities of *computation*, *coordination* and *communication*, we see that for every code base, *coordination* is the largest portion of the code. This lines-of-code count is a preliminary indicator that *coordination* may be the most labour intensive component of code to write. It becomes equally apparent that frameworks like MapReduce and OpenMP work to minimize the amount of work associated with this *coordination*. For example, in these results, MapReduce is the only mechanism in which *task coordination* is smaller than *data coordination* and the *synchronization* is completely implicit.

Synchronization is the smallest piece of *coordination* code in all three implementations. This demonstrates that a lines-of-code count is not the best measure of comprehensive burden, as *synchronization*, even when implicit, can be challenging. We believe these results highlight the complexity that lies in the relationship of *synchronization* to *data coordination* and *task coordination*.

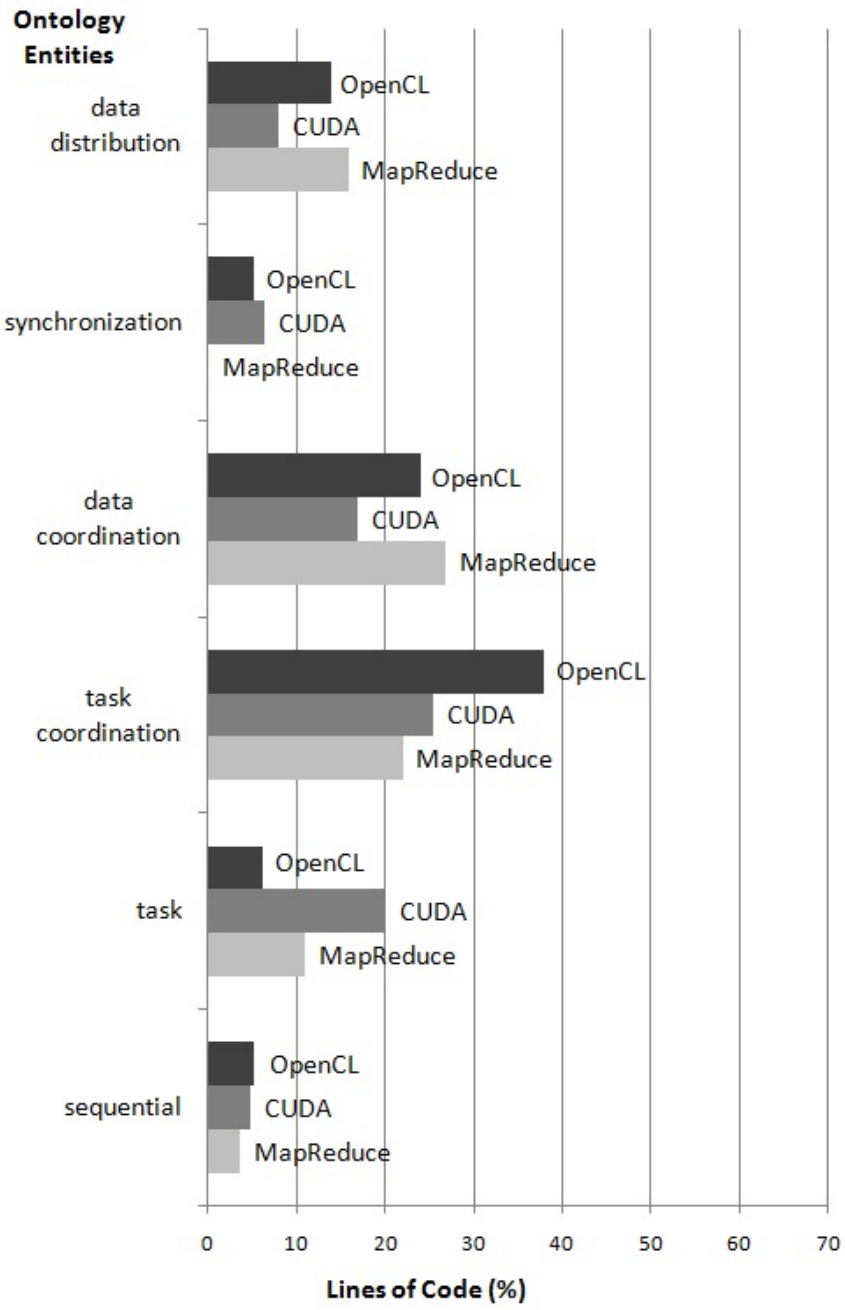

Figure 5.9: Full mapping of OPA onto three reduction implementations

5.3 Mapping OPA to Source Artefacts: Fast Fourier Transform (FFT)

FFT is considered a highly parallelizeable *computation* for which many architecture and input specific implementations exist. The three implementations under analysis here are drawn from the Fastest Fourier Transform in the West (FFTW) (Frigo and Johnson, 2005), a library of highly tuned and performance-centric FFT implementations. This study investigates implementations using POSIX Threads (PThreads) (Institute of Electrical and Electronics Engineers, 2004), OpenMP (OpenMP Architecture Review Board, 2005), and Cell (IBM, 2008) libraries for support. We begin again with an analysis of the PThreads (Figure 5.10), OpenMP (Figure 5.11) and Cell (Figures 5.12 and 5.13) implementations first from a qualitative perspective, mapping OPA entities as they exist in source code, followed by the quantitative results of a full OPA mapping to source. Though this study considers only a small portion of these three FFT implementations, we have chosen to demonstrate the common elements that exist between the three implementations.

5.3.1 Qualitative Analysis of FFT Implementations

Computation: Given that this case study considers only portions of these three code bases, the amount of code designated purely to *computation* is small, even in comparison to the less complex, simple reduction algorithm. Specifically, in this analysis we examine only the breakdown of the *computation* and its dispatch to the functions which perform the actual sub-transform computations.

In the predominant loop of the PThreads implementation, shown in Figure 5.10, a chunk of data is selected and passed on for parallelized *computation* (Figure 5.10: line 16). The OpenMP implementation has a similar loop preceded by a directive: `#pragma omp parallel private(d)` (Figure 5.11: line 4). The OpenMP directive, if enabled, essentially flattens this loop and the *computation* performed on each chunk of data is handled concurrently by individual threads. Though the directive is a catalyst for possible *communication*, from the developer perspective, the whole loop (Figure 5.11: lines 5-11) can be classified as merely *computation*.

The Cell-specific implementation has a similar modular breakdown to that of OpenCL and CUDA in that it provides host code (Figure 5.12) that will run on the Power Processing Unit (PPU), overseeing operations (Figure 5.13) running on each

```
1  void X(spawn_loop)(..)
2  {
3    ...
4
5    for (i = 0; i < nthr; ++i) {
6      struct work *w = &r[i];
7          spawn_data *d = &w->d;
8          d->max = (d->min = i * block_size) + block_size;
9          if (d->max > loopmax)
10           d->max = loopmax;
11         d->thr_num = i;
12         d->data = data;
13         w->proc = proc;
14
15         if (i == nthr - 1) {
16           proc(d);
17         } else {
18           os_sem_init(&w->done);
19
20           WITH_QUEUE_LOCK({
21             if (worker_queue) {
22                     struct worker *q = worker_queue;
23                     worker_queue = q->cdr;
24                     q->w = w;
25                     os_sem_up(&q->ready);
26         } else {
27                     os_create_worker(worker, w);
28                   }
29       });
30     }
31   }
32
33   for (i = 0; i < nthr - 1; ++i) {
34     struct work *w = &r[i];
35         os_sem_down(&w->done);
36         os_sem_destroy(&w->done);
37   }
38 ...
39 }
```

Figure 5.10: FFTW PThread implementation of the `spawn_loop` function.

```
1  void X(spawn_loop)(..)
2  {
3    ...
4  #pragma omp parallel for private(d)
5    for (i = 0; i < nthr; ++i) {
6      d.max = (d.min = i * block_size) + block_size;
7           if (d.max > loopmax)
8              d.max = loopmax;
9           d.thr_num = i;
10          d.data = data;
11          proc(&d);
12   }
13   ...
14 }
```

Figure 5.11: FFTW OpenMP implementation of `spawn_loop` function.

Synergistic Processing Unit (SPU). This FFT SPU code loops through a simple finite state machine, moving through the stages of *computation*, setting the initial context within the PPU code (Figure 5.12: line 9).

Each SPU performs an isolated and identical set of operations on a subset of the data. Each SPUs `main` function (Figure 5.13) loops through a simple finite state machine of four stages: `copy`, `DFT`, `transpose` and `exit`.

Communication: *Communication* in these three implementations is tightly tied to the *computation*. In fact, in the OpenMP implementation, there is no pure *communication* identified beyond the pragma. For this reason, in this section we focus on the PThreads and Cell implementations where we see explicit *communication* in the form of semaphore support in PThreads (Figure 5.10: lines 18, 25, 34-35) and synchronization mechanisms within the SPU module (Figure 5.13: line 4) and the host code (Figure 5.12: lines 41, 44).

Concept overlap – computation/communication: The fine-grained control of PThreads supports local *computation* (Figure 5.10: line 17-29) or the dispatch of that *computation* to a worker (Figure 5.10: line 16). This subtle coupling of *computation* and *communication* can allow for better resource utilization through thread or worker reuse, otherwise, a new worker is created for the current *computation* (Figure 5.10: lines 19-25).

Though the OpenMP implementation lacks the fine-grained explicit control of PThreads, the OpenMP compiler can directly determine the number of threads created and can assign *computation* based on the number of iterations of the loop.

```
1  static void apply(const plan *ego_, R *ri, R *ii, R *ro, R *io)
2  {
3    ...
4    for (i = 0; i < nspe; ++i) {
5      int chunk;
6          struct spu_context *ctx = X(cell_get_ctx)(i);
7          struct dft_context *dft = &ctx->u.dft;
8
9      ctx->op = FFTW_SPE_DFT;
10
11         dft->r = ego->radices;
12         dft->n = ego->n;
13         dft->is_bytes = ego->is * sizeof(R);
14         dft->os_bytes = ego->os * sizeof(R);
15         dft->v[0] = ego->v[0];
16         dft->v[1] = ego->v[1];
17         dft->sign = ego->sign;
18         A(FITS_IN_INT(ego->Wsz * sizeof(R)));
19         dft->Wsz_bytes = ego->Wsz * sizeof(R);
20         dft->W = (uintptr_t)ego->W;
21         dft->xi = (uintptr_t)xi;
22         dft->xo = (uintptr_t)xo;
23
24     /* partition v into pieces of equal size,
25     subject to alignment constraints */
26         if (cutdim == 0 && !contiguous_r) {
27           /* CUTDIM = 0 and the SPU uses transposed DMA.
28           We must preserve the alignment of the dimension 0 in the cut*/
29           chunk = VL * ((v - ego->v[cutdim].n0) / (VL * (nspe - i)));
30         } else {
31           chunk = (v - ego->v[cutdim].n0) / (nspe - i);
32         }
33
34         dft->v[cutdim].n1 = v;
35         v -= chunk;
36         dft->v[cutdim].n0 = v;
37     ...
38     A(v == ego->v[cutdim].n0);
39
40     /* activate spe's */
41     X(cell_spe_awake_all)();
42
43     /* wait for completion */
44     X(cell_spe_wait_all)();
45 }
```

Figure 5.12: FFTW Cell-specific implementation of the PPU control of SPUs.

```
1  int main(unsigned long long spu_id, unsigned long long parm)
2  {
3    ...
4    for (;;) {
5      wait();
6
7          /* obtain context */
8          X(spu_dma1d)(&ctx, parm, sizeof(ctx), MFC_GET_CMD);
9
10     switch (ctx.op) {
11             case FFTW_SPE_DFT:
12                     X(spu_do_dft)(&ctx.u.dft);
13                     break;
14
15             case FFTW_SPE_TRANSPOSE:
16                     X(spu_do_transpose)(&ctx.u.transpose);
17                     break;
18
19             case FFTW_SPE_COPY:
20                     X(spu_do_copy)(&ctx.u.copy);
21                     break;
22
23             case FFTW_SPE_EXIT:
24                     return 0;
25     }
26
27         /* signal completion: */
28         ctx.done = 1;
29         X(spu_dma1d)(&ctx, parm, sizeof(ctx), MFC_PUTF_CMD);
30   }
31 }
```

Figure 5.13: FFTW Cell-specific implementation of SPU workload.

Computation and *communication* do overlap (implicitly) at the `#pragma` from a developer's perspective. The determination of the amount of data to pass to a worker for processing is dependent on the number of workers available.

While the code shown in Figure 5.13 is dedicated to run on the SPUs, it is not pure *computation*. The code segment demonstrates how each SPU must `get/put` the current context (Figure 5.13: lines 3 and 14) through *communication* via shared memory with the PPU. The code then provides communication within a `switch` statement, determining the correct *computation* to be performed based on the current context (Figure 5.13: lines 10-25).

The actual context assignment, placement of data and the initiation of SPU execution occurs at the PPU and is difficult to trace through manual inspection. A portion of this *coordination* of *computation* and *communication* is shown in Figure 5.12. The approach taken in this FFT implementation is similar to that of OpenMP and PThreads, using a loop that iterates through each SPU, setting its operation context and assigning it a portion of that memory that holds the data on which to be operated. The coupling of *communication* and *computation* is demonstrated in the context acquisition required to setup the SPU (Figure 5.12: lines 6-7). Additionally, the *communication* of data is tied to the *computation* to be performed on the data in the Cell architecture; SPUs can only perform *computation* on a contiguous data block (Figure 5.12: line 26-32) and the *communication* of data must take this into account.

5.3.1.1 Summary

Linguistic support is minimal in both PThreads and OpenMP, but for very different reasons. PThreads supports a high level of control in terms of how *computation* is split up and performed, and generic support for *communication* in the form of synchronization constructs. OpenMP handles all of these resource decisions and synchronization in the background, concealing the need for linguistic support. The Cell implementation leverages naming conventions in key identifiers that assist in the identification of *computation* code that would run on the SPUs. Function naming conventions and built-in library mechanisms for memory allocation, reading data and coordinating the *computation*, helps to identify code associated with *communication*.

5.3.2 Quantitative Results of Ontology Mapping

This section provides the quantitative results of a manual line-by-line mapping of OPA to the three FFT implementations in this experiment. Just as in the reduction experiment, this mapping is guided by Table 4.2, which outlines the mapping of OPA entities to source code.

Figure 5.7 shows the percentage of code from each implementation assigned to the *computation* and *communication* entities. Again, for each implementation, the combination of *computation* and *communication* counts exceeds 100% of the code base. Figure 5.8 isolates this intersection between *computation* and *communication*, showing again that the intersection of these two entities is in fact substantially larger than either single entity in isolation for all three examples.

Figures 5.14 and 5.15 show that, in the FFTW code base, this dominant intersection is again exhibited in these implementations. We now plunge deeper into its possible ramifications.

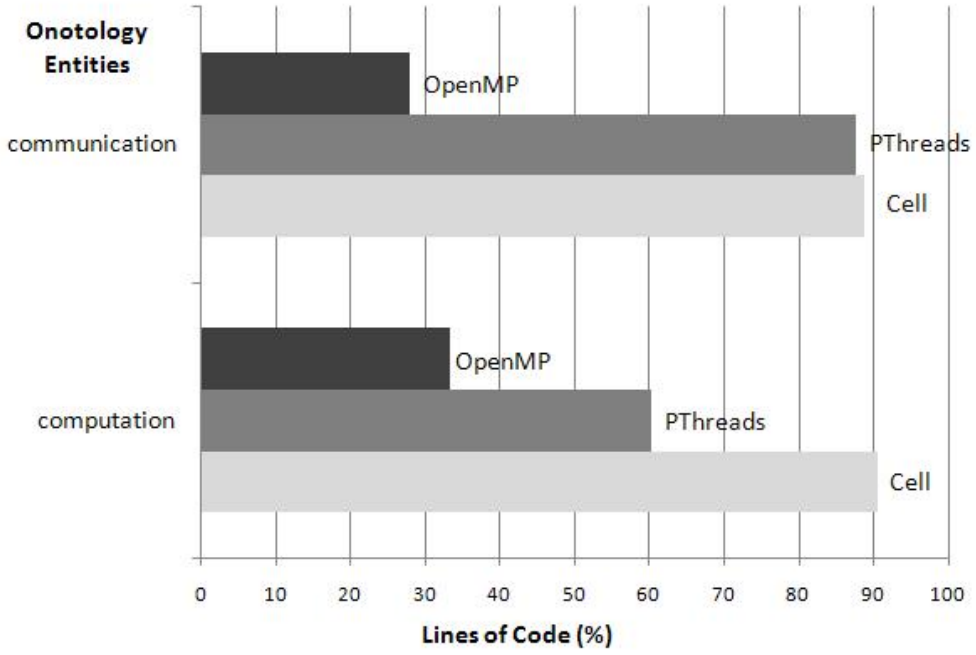

Figure 5.14: *Computation* and *communication* breakdown in three implementations of a FFT

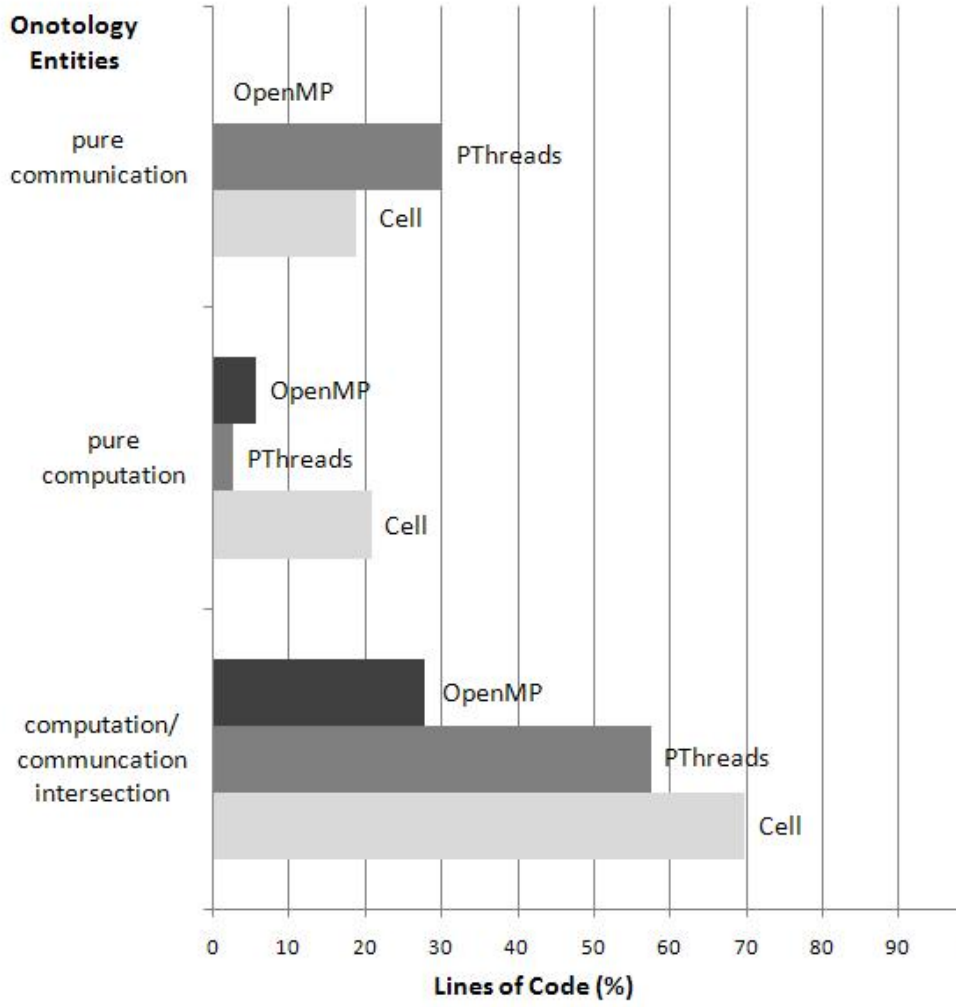

Figure 5.15: *Computation* and *communication* intersection in three implementations of a FFT

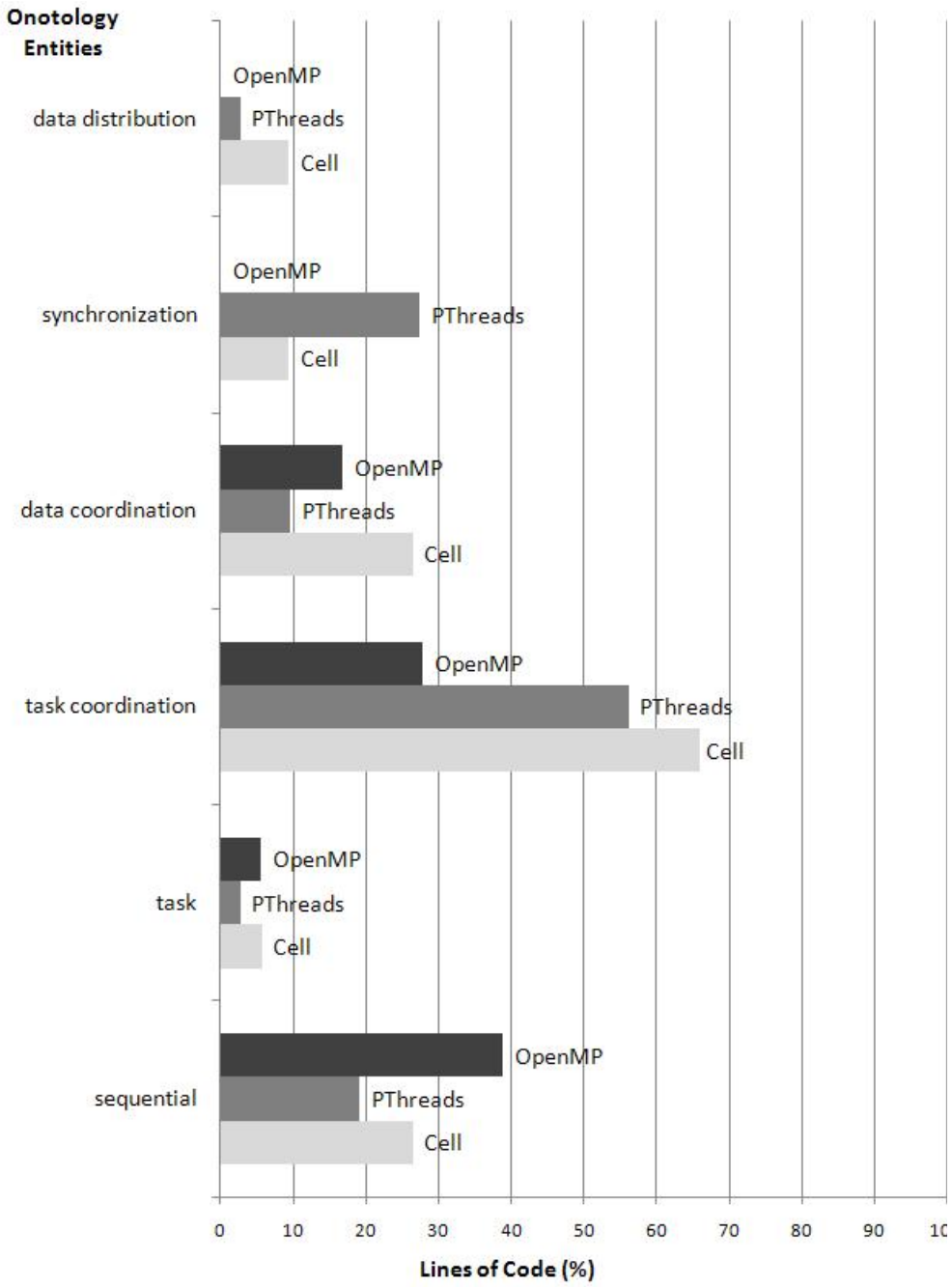

Figure 5.16: Full mapping onto three FFT implementations

Figure 5.16 provides the results of a full mapping of OPA to the three FFT implementations using OpenMP, PThreads and Cell libraries. Similar to the simple reduction code examples, *data distribution* and *task* entities are small in terms of lines-of-code percentages. This result can be attributed to the limited scope of the

code actually being analyzed. Also similar to the results from the reduction case study (Section 5.2), *task coordination* accounts for the largest portion of the code.

OpenMP conceals all *synchronization* and *data distribution* from the developer. This is in contrast to the result we see with the Cell and PThreads implementations, which introduce more than double the percentage of the code base than any other mechanism across both case studies. In addition, the Cell implementation requires more explicit *data coordination* than that of PThreads in order to ensure the data for each worker resides in a contiguous block of memory.

5.4 Mapping to Design Artefacts: The MapReduce Pattern

The MapReduce design pattern is a common parallel design pattern that provides a solution to a problem involving many independent computations that must be collected and merged into one result. The *Pattern Language for Parallel Programs* (Berkeley Parallel Computing Lab, 2010) includes a full description of this pattern with a description of the *problem*, a *context*, the *forces* impacting the implementation and the *solution* both in terms of the general structure and implementation-specific details.

In this case study we are interested in how the design pattern presents itself statically in a given set of implementations. Specifically, we are interested in comparing implementations in terms of the tradeoffs made. For the purpose of this study we extract four *forces* from the design pattern that play out as implementation tradeoffs described in the solution section of the pattern write-up.

The *forces* considered here are: *distribution*, *communication*, *number of tasks* and *load balancing*. Specifically, the pattern describes the need for a good *distribution* to ensure a scalable solution across an increasing number of processing units. This *distribution* may require too much interaction between processing units and in turn impact the *communication force* which strives to minimize overhead. The *number of tasks* a problem is broken down into can also introduce *communication* overhead if the tasks are too fine-grained. A task granularity that is too coarse-grained will in turn be difficult to *load balance* across processing units where *load balancing* should in best case be transparent to the programmer.

Now, with the identification of specific *forces* impacting an implementation of the MapReduce pattern, we can begin to investigate their relationship to the *entities* of OPA. In this exercise we leverage the code colouring of OPA to narrow the code segments for inspection of implementation tradeoffs associated with specific *forces*. Figure 5.17 shows thumbnails of a manual colouring of the code bases to highlight the six low-level entities of OPA. It is important to note that in several cases, a single line of code could be categorized in terms of more than one entity (dual colouring).

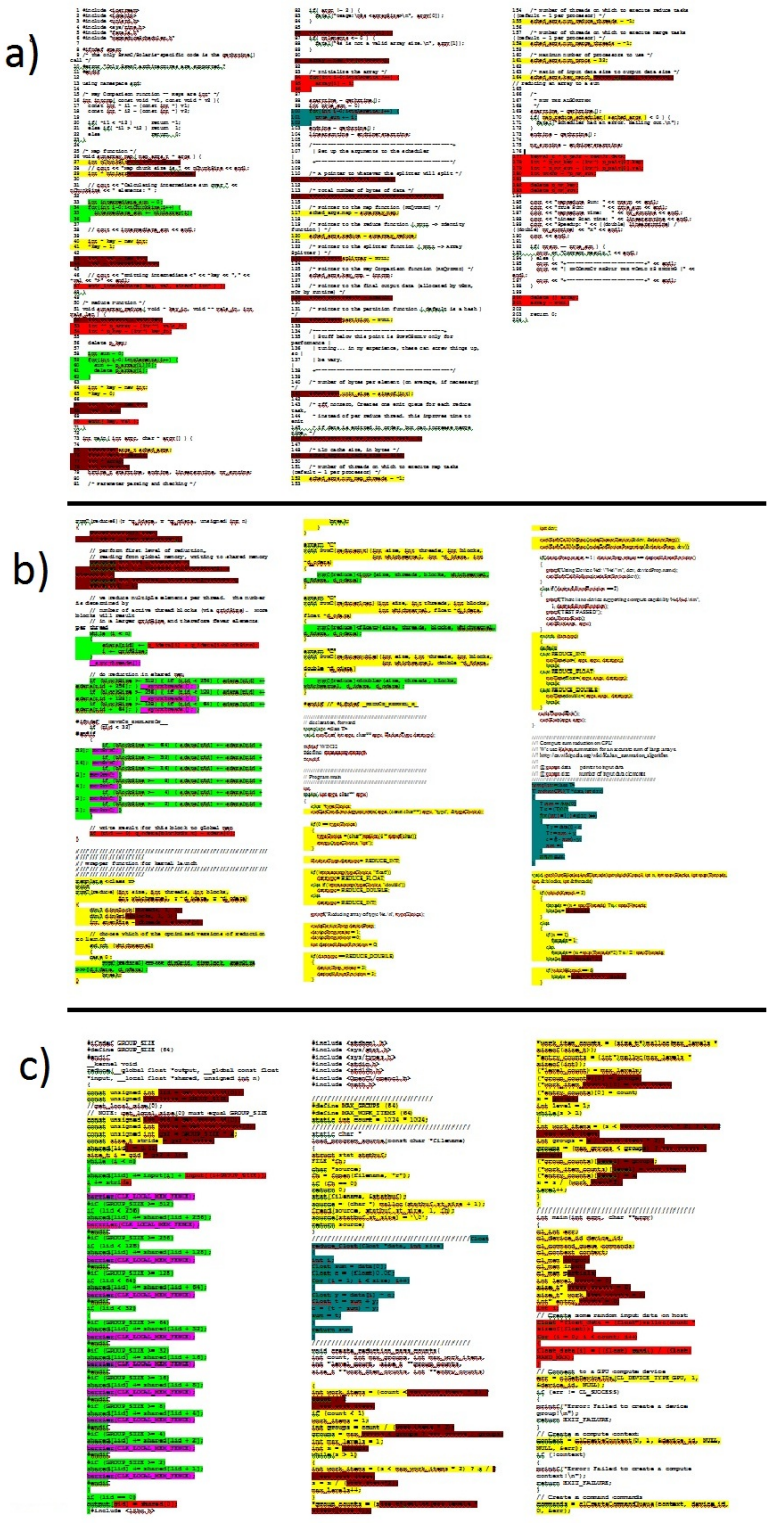

Figure 5.17: Manual colouring of each of the three implementations: (a) MapReduce framework, (b) CUDA and (c) OpenCL kernels

5.4.1 Aligning *Forces* with *Entities*

The first step is to identify which *entities* of the ontology each *force* aligns with. In a preliminary pass we can logically associate the *communication force* with OPA's *communication* entity which breaks down into the low-level data *distribution* and *synchronization* entities (coloured red and pink, respectively).

The other three *forces* we logically associate with low-level *entities* of OPA. Specifically, the *number of tasks force* we associate with the *task coordination entity* as the granularity of tasks is dictated by the code to set up the tasks themselves. Therefore, we can focus our code inspection on just the *task coordination* regions (coloured yellow).

An initial reaction would be to associate the *distribution force* with the *data distribution* entity of OPA, but in fact it is the *data coordination* entity that highlights the code that handles distribution setup. The *data distribution* regions contain the actual transfer of data. Therefore, when investigating the *distribution force* we look in both the *data coordination* and *data distribution* regions (coloured burgundy and red, respectively).

Load balancing is the more challenging force to associate with a single *entity* of the ontology as it is impacts both the *task coordination* and *data coordination* code. Therefore, like *distribution* we must investigate two regions of the ontology: *task coordination* and *data coordination* (coloured yellow and burgundy, respectively).

Table 5.1: Pattern *forces*, with correlated ontology *entities* of OPA and corresponding mapping colours

Pattern Forces	Ontology Entities	Colouring
Distribution	Data Distribution	red
	Data Coordination	burgundy
Number of Tasks	Task Coordination	yellow
Communication	Data Distribution	red
	Synchronization	pink
Load Balancing	Task Coordination	yellow
	Data Coordination	burgundy

Table 5.1 summarizes these associations between pattern *forces* and ontology entities. In this form the extent of the interaction and dependencies between *forces* becomes more evident. Figure 5.18 further provides a graphical representation of

these overlaps suggesting a double dependency of *load balancing* on *distribution* and *number of tasks*, and further the relationship between *communication* and *distribution*.

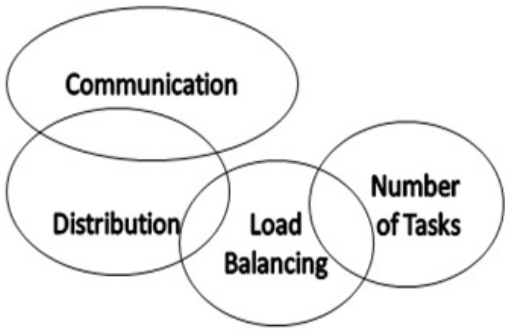

Figure 5.18: Relationship between *forces* in the MapReduce pattern

5.4.2 Mapping *Forces* to Code

The following subsections overview our analysis of three MapReduce implementations based on their alignment between *force* and *entity* relationships summarized in Table 5.1 (Section 5.4.1).

5.4.2.1 Communication

In terms of the *communication* introduced by *synchronization* of access to shared data, the MapReduce framework shields developers from this complexity and ensures safe data access within built-in functions. The OpenCL and CUDA paradigms more closely resemble a C-based language, as explicit *synchronization* calls are required within the computation that will run on the processing unit. Language-specific *synchronization* mechanisms of `barrier` and `syncthreads` functions are called upon to ensure proper access to share the data structure. The CUDA implementation contains additional explicit*synchronization* in comparison to the OpenCL implementation with a call to the `cudaThreadSynchronize` function just before the result is retrieved from the compute units.

This implicit versus explicit *synchronization* characteristic aligns with the lines-of-code counts in Figure 5.9 (Section 5.2) that show the MapReduce framework as

having 0% of the code identified as synchronization compared to 5 to 10% of that of the OpenCL and CUDA code.

5.4.2.2 Distribution

The MapReduce design pattern specifies the importance of a good *distribution* of data in order to provide a scalable solution. Figures 5.19, 5.20 and 5.21 are code listings that contain regions of code that handle setup for *data distribution* in the MapReduce, OpenCL and CUDA implementations, respectively. The setup regions of code within these listings are classified as *data coordination* according to the OPA ontology.

Again, the MapReduce implementation in Figure 5.19 hides much of the complexity by leaving most of the *data distribution* details to the underlying architecture. Lines 13 and 14 demonstrate this with the data splitting left to the default setting.

The OpenCL and CUDA implementations contain a much more involved setup of *data distribution*. Line 2 of the OpenCL code in Figure 5.20 and lines 4 and 5 of the CUDA implementation in Figure 5.21 were marked as *data coordination* regions according to OPA. The tagging of these regions highlights the similar ways these implementations create an input buffer, but also shows the differences in the linguistic support with OpenCL's higher level function `clCreateBuffer`, as opposed to CUDA's C-like `cudaMalloc` function.

5.4.2.3 Communication/Distribution Overlap

The previous two subsections describe *communication* in the form of *synchronization* and *coordination* code necessary for *data distribution*. The other entity that relates to both *communication* and *distribution* is the actual *data distribution* itself. The placement of a result into a shared data structure can be considered an implicit form of *communication* as a signal to cooperating compute units, while at the same time it performs the job of the actual explicit *data distribution*.

Lines 1 through 4 of Figure 5.19, lines 3 and 4 of Figure 5.20 and lines 6 through 8 of Figure 5.21 are classified as *data distribution* according to OPA. These marked regions again highlight the strong underlying infrastructural support of MapReduce, leaving little to the programmer in comparison to OpenCL and CUDA.

```
1  /* Initialize the array */
2  for(int i=0;i<nElements;i++) {
3        array[i] = i;
4  }
5  /* A pointer to whatever the splitter will split */
6  sched_args.task_data = array;
7
8  /* Total number of bytes of data */
9  sched_args.data_size = nElements*sizeof(int);
10
11 /* Pointer to the map function (REQUIRED) */
12 sched_args.map = sumarray_map;
13
14 /* Pointer to the reduce function
15 ( NULL => Identity function ) */
16 sched_args.reduce = sumarray_reduce;
17
18 /* Pointer to the splitter function
19 ( NULL => Array Splitter ) */
20 sched_args.splitter = NULL;
```

Figure 5.19: MapReduce distribution

```
1  // Create the input buffer on the device
2  input = clCreateBuffer(context, CL_MEM_READ_WRITE,
3            sizeof(float) * count, NULL, NULL);
4
5  //Fill input buffer with host allocated random data
6  err = clEnqueueWriteBuffer(commands, input, CL_TRUE, 0,
7            sizeof(float) * count,(void *)float_data,0, NULL, NULL);
```

Figure 5.20: OpenCL distribution

```
1  // allocate device memory and data
2  T* d_idata = NULL;
3  T* d_odata = NULL;
4  cutilSafeCallNoSync( cudaMalloc((void**) &d_idata, bytes) );
5  cutilSafeCallNoSync( cudaMalloc((void**) &d_odata, gnumBlocks*sizeof(T)));
6
7  // copy data directly to device memory
8  cutilSafeCallNoSync( cudaMemcpy(d_idata, h_idata, bytes,
9                           cudaMemcpyHostToDevice) );
10 cutilSafeCallNoSync( cudaMemcpy(d_odata, h_idata,
11                           numBlocks*sizeof(T), cudaMemcpyHostToDevice));
```

Figure 5.21: CUDA distribution

5.4.2.4 Number of Tasks and Load Balancing

The *task coordination* marked regions within the three code bases in question contain code that deals with all the necessary task setup code for tasks. But, as witnessed in both the simple reduction (Section 5.2) and the FFT case study (Section 5.3), some lines of code cannot be mapped to just one entity in the application of the lower-level of the ontology. One of the common overlaps in the three code bases in question is between *data coordination* and *task coordination* code. That is, some of the lines of code are marked as both *data coordination* and *task coordination* regions.

As has been demonstrated so far, the OpenCL and CUDA implementations require more in terms of *coordination* setup, in turn giving the programmer more control over issues like *load balancing*. Functions like `create_reduction_pass_counts` from the OpenCL code and `getNumBlocksAndThreads` from the CUDA code base regions that demonstrate this dual responsibility. The CUDA function shown in Figure 5.22 contains lines of code that determine thread allocation through task decomposition and data partition sizes in terms of blocks. This tight coupling between the *distribu-*

tion, the *number of tasks*, and the number of threads are all key to balancing a load in a parallel application.

Considering the *task coordination* and *data coordination* in coloured regions in the MapReduce implementation, the tension of the *load balancing force* becomes visible in the scheduling argument assignments. The control over *load balancing* is not as fine-grained in the MapReduce implementation as in the CUDA and OpenCL implementations but these argument settings work together to tune the application for performance and resource utilization.

```
1  if (whichKernel < 3) {
2     threads = (n < maxThreads) ? n : maxThreads;
3     blocks = n / threads;
4  }
5  else
6  {
7     if (n == 1)
8        threads = 1;
9     else
10       threads = (n < maxThreads*2) ? n / 2 : maxThreads;
11    blocks = n / (threads * 2);
12
13    if (whichKernel == 6)
14       blocks = min(maxBlocks, blocks);
15 }
```

Figure 5.22: CUDA *task* and *data coordination*

5.5 Summary

In order to address objective O5:

- **O5.** Use the conceptual model developed to describe and map to multiple artefacts in a uniform way.

this chapter provides a case study mapping OPA to both low-level source artefacts and higher-level design artefacts to evaluate the applicability of OPA to multiple artefacts. Further, this case study identified ways in which OPA can commonly apply and link different artefact representations of a single solution.

This evaluation of OPA in the context of Rupture software artefacts included its full mapping on to six code bases, demonstrating the feasibility and fit of the

ontology to this domain. The mapping of OPA to patterns not only further confirms its applicability within this domain, but begins to consider OPA as a conceptual link between the various representations of a problem and solution. That is, the mapping of the ontology to patterns through to source demonstrates that this form of abstraction could provide a possible bridge between various levels of representation of a parallel solution, from description to design to code.

The mapping of OPA onto multiple artefact types includes: pedagogical activities, source code and design patterns introduces the ability to transfer the mental model of parallel issues at the level of pedagogical activities through to patterns and further to source code and other software artefacts. Future work will consider this approach for pedagogical purposes and will investigate the mapping of the ontology to the other software artefacts discussed in Chapter 2.

Here, we identify the following contributions covered in this chapter that address the thesis of this work:

- Determined how the proposed ontology maps to low-level, source artefacts.
- Determined how the proposed ontology maps to design artefacts.
- Documented opportunities for tying design artefacts to source artefacts.
- Used the documented opportunities to identify a uniform application of the ontology across a variety of software artefacts.

The following chapter evaluates the proposed Rupture model in the context of a framework to provide tool support for the analysis of Rupture artefacts.

Chapter 6

Tools to Support the Rupture Model

This chapter leverages RIPL (Chapter 3) and OPA (Chapter 5) to evaluate the proposed Rupture model for tracking change artefacts to support the development of parallel software. Targeting research objective **O6**, **O7** and **O8** of the thesis, this chapter proposes *C_3PO*, a framework to support the Rupture model in the analysis of change artefacts. We evaluate this proposed tool support in the context of Rupture artefact examples both at abstract and implementation levels and finally demonstrate its applicability in the context of existing optimization processes. In support of these objectives, this chapter specifically addresses the following research question:

> *How can the proposed support models of this dissertation be integrated through tools to support the parallel development process?*

6.1 Introduction

Workflows identified in Rupture (Chapter 2) to support the parallel development process must link changes in dynamic properties of a system back to the tracked changes in static artefacts. As a result, a consistent way of viewing changes and navigating between them is necessary. Figure 6.1 outlines the customizable framework we call C_3PO to support the visualization of *Computation, Communication and Coordination for Parallel Objectives*. Each of the six change sets to be visualized with this framework is drawn from the Rupture model and should be supported by an interchangeable, and potentially language and domain-specific tool for visualizing

change artefacts. The integration of the visual representation of the artefacts would be handled by the higher-level framework of C_3PO.

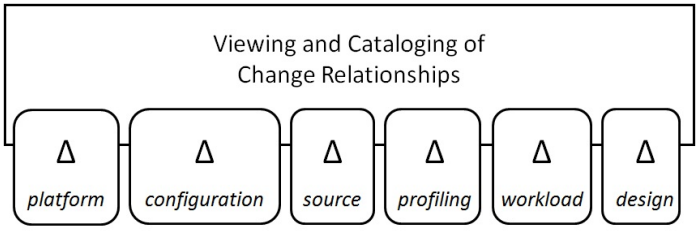

Figure 6.1: Structure of proposed framework with customizable extensions

Unification of the views to a common format allows for a common starting point to iterate on feedback loops involving dynamic characteristics and a series of causally related changes over time. Support for more global reasoning can be provided by saving system snapshots; that is, the representation of the current state for each artefact that collectively defines a parallel application: *design*, *platform*, *source*, *configuration*, *profiling*, *workload*, and *dynamic* characteristics.

Though we envision policies to automate snapshot support, perhaps triggered by time or number of changes, our own prototype tools currently rely on manual practices to identify semantically-significant points in development, and a standard version control system for storage. Figure 6.2 highlights the ways in which the relative differences ($\Delta_{i,j}$) between two snapshots (SS_i, SS_j) can be acquired. Integration of these tools should not only include support for navigation from performance results to static changes, but also a comprehensive means of tracking correlated static and dynamic changes over time, via system-snapshots.

The framework provides a language-agnostic, semi-automated, general-purpose process for creating and visualizing performance results that are navigable to a parallel code base. This preliminary tool suite includes: (1) visualisation of static changes at multiple levels of abstraction, (2) performance profiling of dynamic changes and (3) an integration of static and dynamic views. In the remainder of this chapter, an evaluation of the proposed C_3PO framework as support for the Rupture model is provided at the granularity of the OPA ontology (Section 6.2) and lines-of-code (Section 6.3). A case study evaluating the efficacy of the Rupture model in terms of existing parallel optimization workflows (Section 6.4) is followed by a summary and concluding remarks (Section 6.5).

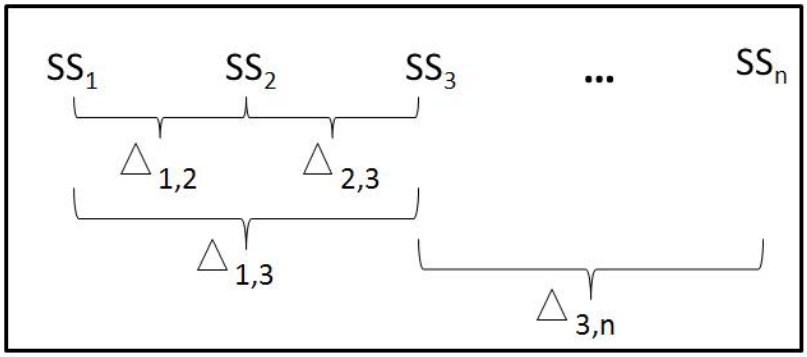

Figure 6.2: Snapshots (SS_i) of all system artifacts can be catalogued at any time, promoting further reasoning support for changes ($\Delta_{i,j}$) between any two snapshots (SS_i, SS_j)

6.2 An Ontology Perspective

While the generalized ontology of OPA described in Chapter 4 provides useful domain knowledge in the form of entities and relationships that make up a parallel application, without the ability to link this abstraction to a code base it lacks usefulness in terms of integration with software development life cycles. We consider the ways in which new and existing tools for OPA can be leveraged to integrate with and support the Rupture model (Section 6.2.1), followed by a preliminary user study, providing an initial evaluation from a user perspective and insights in terms of value in code navigation as well as barriers to use (Section 6.2.2).

6.2.1 Supporting the Rupture Model

To create a navigable code base we must be able to map the ontology to an actual implementation instance providing a view in terms of OPA entities that supports the Rupture model through an abstracted view of change artefacts and navigation to a narrowed source view. While the ability to narrow a developer's view facilitates navigation and comprehension, we believe OPA can provide further assistance in terms of establishing an explicit representation of the relationships between entities that are typical causes of programming pitfalls. Subsequently, we investigate the feasibility of identifying and linking properties and relationships to implementation in order to leverage existing ontology tools for reasoning and analysis from the OPA level of abstraction down to source.

In order to integrate with and support the Rupture model, we propose an extension to C_3PO leveraging OPA for visualization of a code base at the level of entities. Figures 6.3 and 6.4 demonstrate the file-level OPA perspective on the three implementations of a simple reduction as described in Chapter 5. The horizontally distributed, vertical, white boxes represent individual source files and the coloured lines within the boxes represent the lines of code categorized in terms of OPA entities. Figure 6.3 is coloured based on the three coarse-grained entities of OPA, while Figure 6.4 visualizes the source files in terms of the six the fine-grained entities.

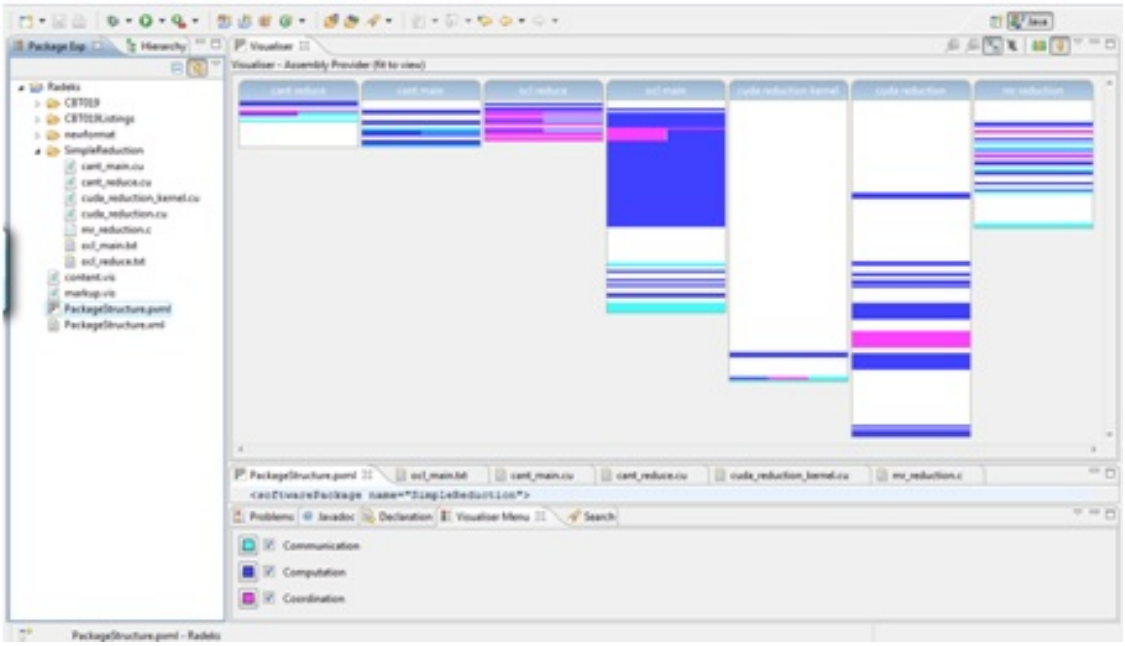

Figure 6.3: C_3PO high-level concern perspective

The five files visualized in Figures 6.3 and 6.4 are from the MapReduce, CUDA and OpenCL reduction implementations. The CUDA and OpenCL implementations are comprised of two files in order to separate out the kernel files, while the MapReduce implementation is contained in a single file. This bird's eye view facilitates a high level investigation of Rupture artefacts, specifically to support: 1) viewing large scale changes such as that to *platform* and *design* and 2) narrow the view on slight changes to *source*.

This view within the C_3PO proof-of-concept framework, demonstrates the magnitude and placement of the different entities of OPA across the three reduction implementations. We envision the ability to semi-automate this traceability of con-

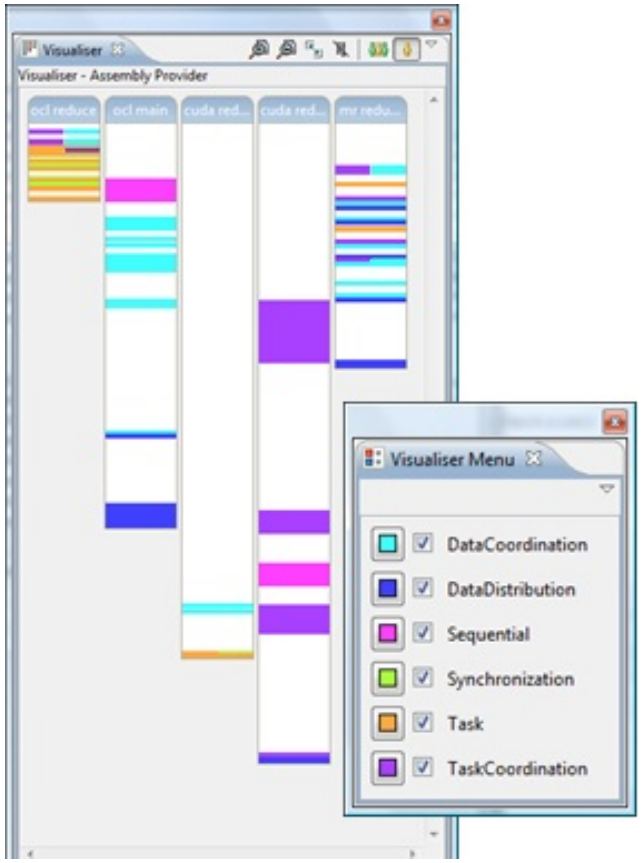

Figure 6.4: Mapping mechanism to OPA entities to establish the feasibility of automation of mapping process

cerns down to lines of code, requiring a mapping of language-specific mechanisms to both high and low-level entities of OPA. Table 6.1 shows a first pass at this mapping, which provides grounds for us to consider the feasibility of the partial automation of this process.

This table provides an overview of how OPA's fine-grained entities are represented within each parallel language. This view shown in Figure 6.3 highlights the similarities and differences at the level of concerns across language mechanisms. The sequential portion of the code is implicit across the board. Similarly, task and data distribution are highly algorithm and implementation dependent and therefore, the support provided by most parallel mechanisms is implicit. Though a thorough comparison of mechanisms is out of the scope of this thesis, it is interesting to note that newer mechanisms in OpenCL are moving towards a more explicit identification of core computation and support for reading/writing to memory.

The coordination of the task and data is also better supported through library functions. For example, OpenCL's support for device setup and queue management far exceeds that provided by CUDA. MapReduce also provides task and data related scheduling parameters for tuning, but the support to make full use of these parameters is not intuitive in our experience. Of the parallel mechanisms that require synchronization to be dealt with by the developer, they each provide some form of linguistic support in the form of locking and timing mechanisms.

Leveraging the mapping discussed above, coupled with user interactive assistance during analysis, we suggest that it may be possible to semi-automate the mapping of the fine-grained ontology to source allowing developers to reason in this way about changes to *platform*. For the development of the C_3PO prototype, however, all code categorization was performed manually, and transformed into an XML file for visualization support.

Table 6.1: Fine-grained concern mapping to language mechanisms

Flag Type	ia64	ibm	ibm64	omni	pgi	sgi	sgi64	sun	sun64
Parallel C									
CC	ecc	xlc_r	xlf_r -q64	ompcc	pgcc pgcc	cc	cc -64	cc	cc
CFLAGS	-O3 -openmp	-O3 -qsmp=omp	-O3 -qsmp=omp	-xO4 -fast	-O3	-O3 -mp	-O3 -mp	-fast -xopenmp	-fast -xopenmp -xarch=native64
C_INC	null	null	null	null	null	null	null	null	null
CLINK	$(CC)	$(CC)	$(CC)	$(CC)	$(CC)	$(CC)	$(CC)	$(CC)	$(CC)
CLINK FLAGS	-O3 -openmp	-O3 -qsmp=omp	-O3 -qsmp=omp	-xO4 -fast	-O3	-O3 -mp	-O3 -mp	-fast -xopenmp	-fast -xopenmp -xarch=native64
C_LIB	null	null	null	null	null	null	null	null	null
Utilities C									
UCC	ecc	cc	cc	cc	cc	cc	cc -64	cc	cc
BINDR	../bin	../bin	../bin	../bin	../bin	../bin	../bin	../bin	../bin
RAND	randi8	randi8	randi8	randi8	randi8	randi8	randdp	randi8	randi8
WTIME	wtime.c	wtime.c	wtime.c	wtime.c	wtime.c	wtime.c	wtime_sgi_64.c	wtime.c	wtime.c
MACHINE	n/a	-DIBM	-DIBM	n/a	n/a	n/a	n/a	n/a	n/a

In terms of supporting Rupture through visualization at the level of *source* artefacts, C_3PO allows a developer to investigate the impact of changes through reasoning at the level of OPA's coarse and fine grained entities and direct navigation to the source code representation. Allowing a developer to quickly identify the parts of an implementation such as *task* or *task coordination* that may introduce *dynamic* overhead to an application and narrow the view to investigate changes in those code segments.

Further, the ability to reason about the more subtle rules for relationships between OPA entities at the level of code will provide crucial reasoning support for the parallel developer. While the visualization support provides entity identification within a code base, it does not express the relationships between these concerns. Users not only need to be able to trace entities to a software artefact, but they must also be able to examine the relationships between entities. In addition to the general relationships between concerns illustrated, the implicit rules and dependencies between entities that must be considered need to be more accessible and explicit.

In an answer to this, we leverage the power of the proposed structure of OPA to not only provide the instances of the entities within a parallel application as a knowledge base, but express the rules and relationships between these entities. These include both the `isA` type relationships between the high and low-level entities of the OPA hierarchy and the the more subtle relationships and rules between entity pairs.

A formalized representation of relationships would buy C_3PO the ability to link into ontology visualization and reasoning tools such as Chimaera (McGuiness *et al.*, 2000). The source code is then both a navigable structure of entities and a queryable instance database, in which queries would reference specific lines of code to further facilitate code navigation. While OPA is now a proof-of-concept ontology for parallel applications and in the development stage, in its completion we envision it would have a full set of properties and restrictions associated with its entities. Table 6.2 provides some examples of the relationships encountered in our parallel code base analysis that would benefit from being made more explicit.

Consider for example the *synchronization* column within Table 6.2. *Synchronization* is a small portion of a code base in terms of a lines-of-code count, but task correctness requires accurate *synchronization* of access to shared data. Further, the same column of Table 6.2 demonstrates that these relationships do not impact just these two concerns. In fact, *synchronization* of data access can force sequential execution of a *task*. The implicit nature of these relationships as they play out in existing

Table 6.2: Concern relationships

	Task (TSK)	Task Coordination (TC)	Data Coordination (DC)	Synchronization (SYN)	Data Distribution (DD)
SEQ	*TSK* is *SEQ* when *SYN*	a $\subset$ of *TC* is *SEQ*	a $\subset$ of *DC* is *SEQ*	*SYN* enforces *SEQ*	*DD* by host is *SEQ*
TSK		*TSK* depends on *TC* *TC* $\subseteq$ *T*	*TSK* depends on *DC* *TSK* $\subseteq$ *DC*	if shared data *SYN* of *TSK* is required	symmetry through *TC* and *DC*
TC			symmetry required	disparate	dependent on balance between *TC* and *DC*
DC				if shared data *SYN* of *DC* is required	*DD* depends on *DC*
SYN					if shared data *SYN* of *DD* is required

parallel language mechanisms could possibly be made more explicit as properties and restrictions within C_3PO's ontology perspective. Table 6.2 provides a small subset of rules as a proof-of-concept, whereas its full form would list all concerns and contain rules for each concern-pair combination.

The visibility of simple dependencies between concerns within code, such as the dependencies of *task* on *task coordination* and *data distribution* on *data coordination* could aide in the comparison of programming language management of these sometimes subtle dependencies. These intricate and often error prone concern relationships impact program accuracy and are handled in a range of different ways across programming languages.

6.2.2 Preliminary User Study

A preliminary user study was performed in order to evaluate the initial applicability of a categorization and marking of lines of code based on general ontology entities. In this user study, four participants were given a set of pre-study questions, three exercises to perform on multiple code bases and a set of post-study questions. The pre-study questions were designed to gather information on experience, while the post-study questions assembled reflection information on the exercises.

Participants performed change tasks on the full implementations of the three simple reductions described in Chapter 5. Each participant was given two printed versions of each implementation, one with the code coloured according to the high-level and

one according to the low-level ontology entities. The participants were asked to complete a set of programming related tasks on paper only, using either the high-level or low-level coloured code or both. The exercises were as follows:

E1. *What would you have to change to increase the number of values added up in the simple reduction implementation?*

E2. *What would you have to change to increase or decrease the number of computation units in the implementation?*

E3. *If the result after running your application was missing the addition of part of the data set, what would your debugging process look like?*

The range in level of general programming expertise of users in this study ranges from undergraduate to graduate level. In addition to this background information, the pre-study survey revealed that the four users had little to no experience with the parallel programming languages considered in this study.

Table 6.3 overviews some of the key results of this user study. We provide results from only Exercise 3 to give an idea of how the OPA mapping was used by each participant. We found that all participants leveraged the code coloring identify the points of interest for debugging purposes, though there was variation in which perspective they used. Specifically, Users A, B and D all looked in *task* code in the low-level perspective, while User C looked in the *computation* code from the high-level.

Table 6.3: User responses to use case questions *Scale: 1-5, where 1 is worse, 3 is neutral, 5 is enhance

User	User's Process for Exercise 3	Ontology Impact*	Preferred Perspective	Qualitative Responses
A	focused on *task* code for placing debug statements	4	low-level	'I was looking for *task coordination* to help with exercise 2 and *task* for exercise 3'
B	iterative debug process investigate *task* code ensure data for sum is 'properly coordinated'	4	mainly high-level	'need time to get used to it(the ontology)'
C	debug statements on data access within *computation* code ensure data for sum is 'properly coordinated'	3.5	high-level	'Once I determined the appropriate colour, I was more likely to ignore the other ones' 'if I picked the wrong colour the the ontology would make things worse' 'the colours helped me categorize what I was looking for'
C	add debug statements to all *task* code	3.5	low-level	'made it easier to locate areas where synchronization could be an issue and places where the computations were being performed'

In general, in this preliminary study participants found the ontology to have a moderately positive impact on code comprehension, but this required a certain level

of trust in the ontology as indicated by the qualitative responses from Users B and C. Qualitative responses like 'made it easier to locate areas' and 'the colours helped me categorize' lead us to believe this visualization coupled with an ontology reasoner engine could achieve the goals of providing a concern based view of source code, and that comprehension tasks could be aided by this tool.

6.3 A Lines-of-Code Perspective

In order to evaluate Rupture in the context of our C_3PO framework at the level of code, we consider the efficacy of the visualization of a wider range of change artefacts and integration within the Rupture workflow. Tools to support the visualization of Rupture's static artefacts including configuration, profiling and source (Section 6.3.1) is followed by integration of the visualization of static and dynamic artefacts to support Rupture's iterative feedback model (Section 6.3.2).

In this preliminary evaluation of the Rupture model and C_3PO, we leverage the benchmark suite code introduced in Section 2.3. Multiple existing tools are leveraged as extensions to the C_3PO framework to visualize the different forms of static and dynamic change artefacts identified in the Rupture model.

For visualisation of static changes we leverage the well established Eclipse plugin, C/C++ Development Toolkit (CDT) (Eclipse, 2010a), which provides IDE support for C/C++ developers. CDT's integration with typical C/C++ tools such as the GNU C++ compiler (g++) (GNU Project, 2010a) and the GNU Project Debugger (GDB) (GNU Project, 2010b) makes it an accepting base for this prototype tool suite. The `compare with` functionality of CDT supports comparison of text files within a workspace or across external folders and provides the support required for static change collection.

Dynamic changes can be captured through performance profiling and resource monitoring. Performance profiling can be platform specific and can include a wide-range of options such as on-line and off-line modes and tracing capabilities. It is key to C_3PO that the results, even system resource utilization (core usage and memory), be made part of a system-snapshot in a consistent manner.

In our prototype tool we leverage the OpenMP specific profiling capabilities of ompP (Furlinger *et al.*, 2005) to capture dynamic changes. This tool uses automatic source code instrumentation of `#pragma` tagged parallel regions, collects profiling information at execution time and reports results in detail. This language specific pro-

filing tool provides a detailed performance breakdown for each parallel region within an application. Each portion of code preceded by a `#pragma omp` is considered a *region* with ompP reporting the region type and location. Timing data and execution counts are provided on a per thread basis with a summation of all thread statistics at the end.

To keep the C_3PO framework language and architecture agnostic, other profiling tools could be used within the framework, but it is important to note that not all profilers take into account parallel execution. For example, the GNU profiler gprof (Fenlason and Stallman, 1998), flattens and sequentializes the otherwise parallel executable in the compilation process and the result is not useful for the verification of performance impact of changes to parallel applications.

6.3.1 Static Artefacts

A consistent way to view static changes in configuration files, source files and source embedded profiling is a high-level perspective, supported through colour-coded differences within an Eclipse viewer. For example, Figure 6.5 provides a visualization of the three 64-bit, architecture-specific configuration files for OpenMP benchmarks with coloured changes relative to the template configuration file. This view highlights the differences between the three configuration files and indicates that in each case (`ia64`, `ibm64`, `sgi64`), the changes to the template were made in roughly the same places.

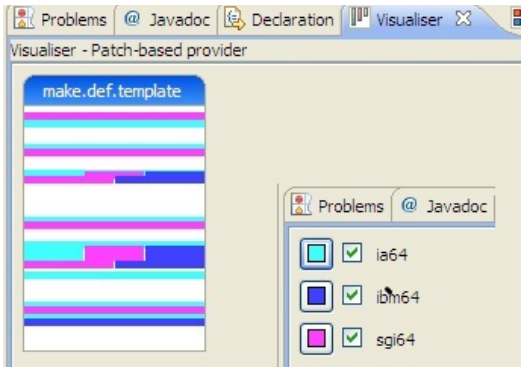

Figure 6.5: Visualisation of Δ*configuration* in NPBs

From such a view, Figure 6.6 demonstrates navigation to the corresponding source view of these fine-grained differences. Further, gutter annotations indicate the places in which a specific configuration differs from the configuration template. The image

shows the specific configuration differences of the `ia64` architecture, with the others tiled behind.

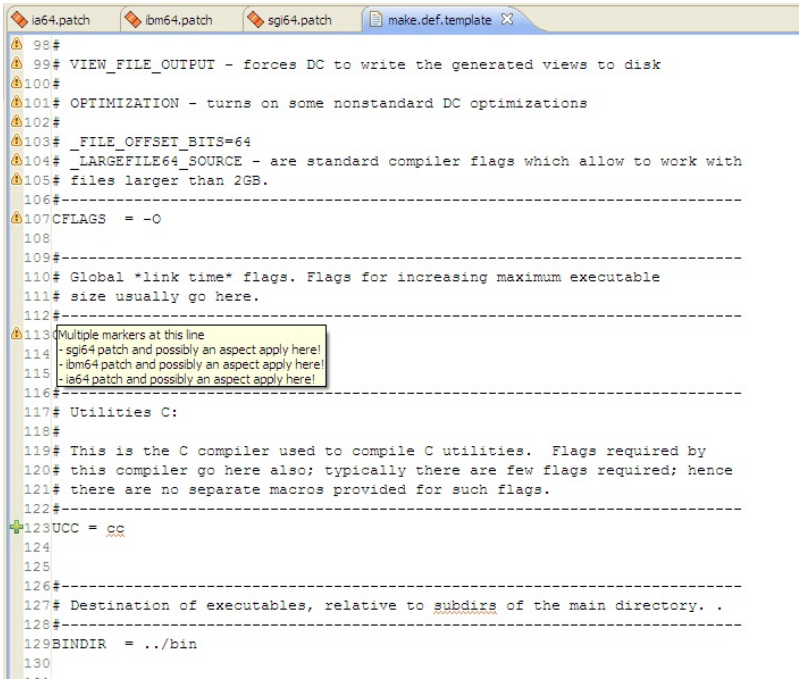

Figure 6.6: Fine-grained flag configuration view for ia64

In this figure, simple flags control the different options for each architecture. This navigation facilitates a developer's ability to experiment with different combinations of flag settings. The importance of this type of experimentation is alluded to in the OpenMP `README` file which states:

`For some buggy OpenMP compilers, you may have to play with the optimization flag, for instance, use "-O2" over "-O3".`

Similarly, Figure 6.7 provides a high-level perspective where parallelization is introduced by both OpenMP (dark) and MPI (light) with respect to the serial implementation. This view shows the more substantial impact of the MPI implementation as well as the overlap in changes across the two implementations. As demonstrated in Figure 6.6, direct navigation to the source for the examination of implementation details provides an integrated view of changes across versions.

Finally, Figure 6.8 provides the visualization of a naïve implementation of profiling which builds on existing profiling support provided by the `TIMING_ENABLED` flag within the NPB-IS benchmark. The implementation introduces two levels of profiling: `PROFILE_ENTER` and `PROFILE_LEAVE`, where the first traces entry to a function

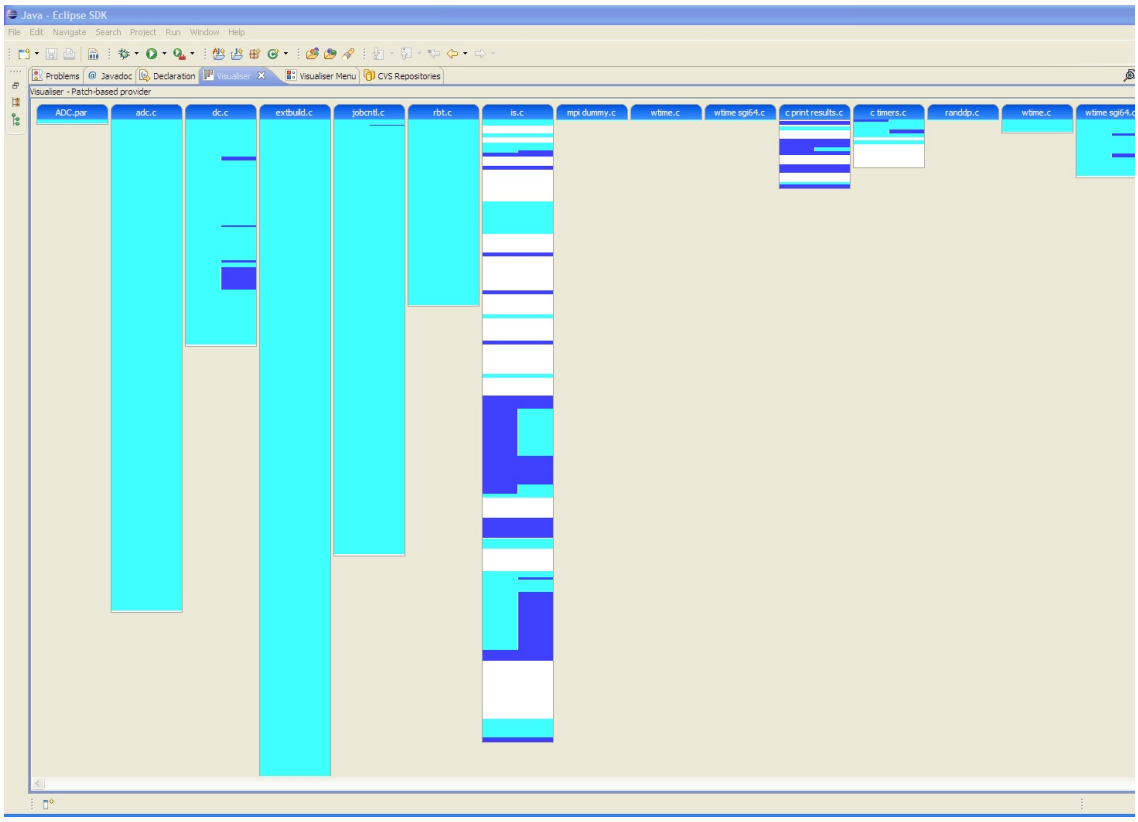

Figure 6.7: Visualisation of $\Delta source$ in NPBs

and a combination of the two trace function entry and exit. The visualizer integrates a high-level view of profiling as it applies to the base system, as well as the ability to configure the profiling through navigation to the editor view.

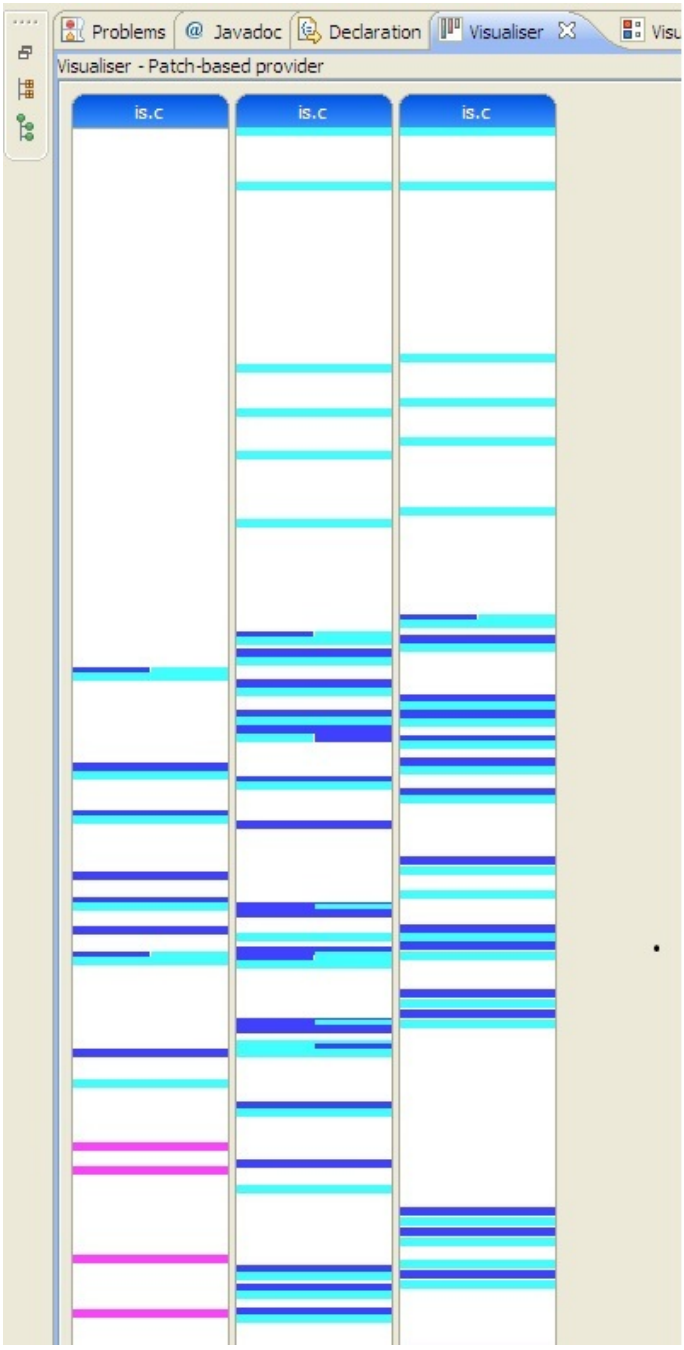

Figure 6.8: Visualisation of Δ*profiling* in NBPs

6.3.2 Integrating Visualizations with Dynamic Artefacts

To demonstrate the integration of static and dynamic deltas (Δs), we provide a proof-of-concept example, leveraging our previous work with tools applied to the parallel domain. In this example, we focus on the molecular dynamic, scientific application (`c_md.c`) from the OmpSCR benchmark suite (Dorta *et al.*, 2005) which contains two parallelizeable regions.

Our previous work with tools (Singh *et al.*, 2007) focused on support for reasoning about conditionally compiled source code based on a set of user-specified conditional flag settings. This initial work provided source code views with dead code either grayed out or folded out of view within the Eclipse IDE viewer (Eclipse, 2010b). This functionality supports the ability to experimentally and iteratively refine configuration settings through changes in flag assignments.

By merging the source visualisation capabilities of Eclipse with performance capturing capabilities of the OpenMP Profiler, ompP (Furlinger *et al.*, 2005) for collection of dynamic data, we further support workflows according to the Rupture model. This integration is facilitated by an automatic parsing of both the source code and its corresponding performance data. A graph of the ompP output, as shown in Figure 6.10, provides a visualization of the dynamic differences between multiple configurations. Each bar in the graph represents the ompP output of a single configuration, corresponding to the Eclipse project the source code resides in.

In this example, a C application whose source is a single file, `c_md.c`, has four different versions. Each version contains a region that supports parallelization defined by *pragmas*. Each of the four versions are contained in separate C projects in Eclipse. Clicking on one of the bars in this chart will open the corresponding version of `c_md.c` within the C editor and will highlight the first parallel region as identified by `#pragma` parser. This Eclipse view, shown in Figure 6.9, allows a user to reason about which region-changes may be responsible for a causal change in performance. In this figure the two versions of the source code that correspond to the `FullParallel` and `TimeParallel` performance entries in the bar chart are shown tiled within the Eclipse view.

These tools are intended to encourage and support developers in the adoption of structured and systematic workflows to analyze how different versions of parallel source code impact the performance of an application. Figure 6.10 demonstrates a navigable view of four different versions of the molecular dynamic application (`c_md.c`)

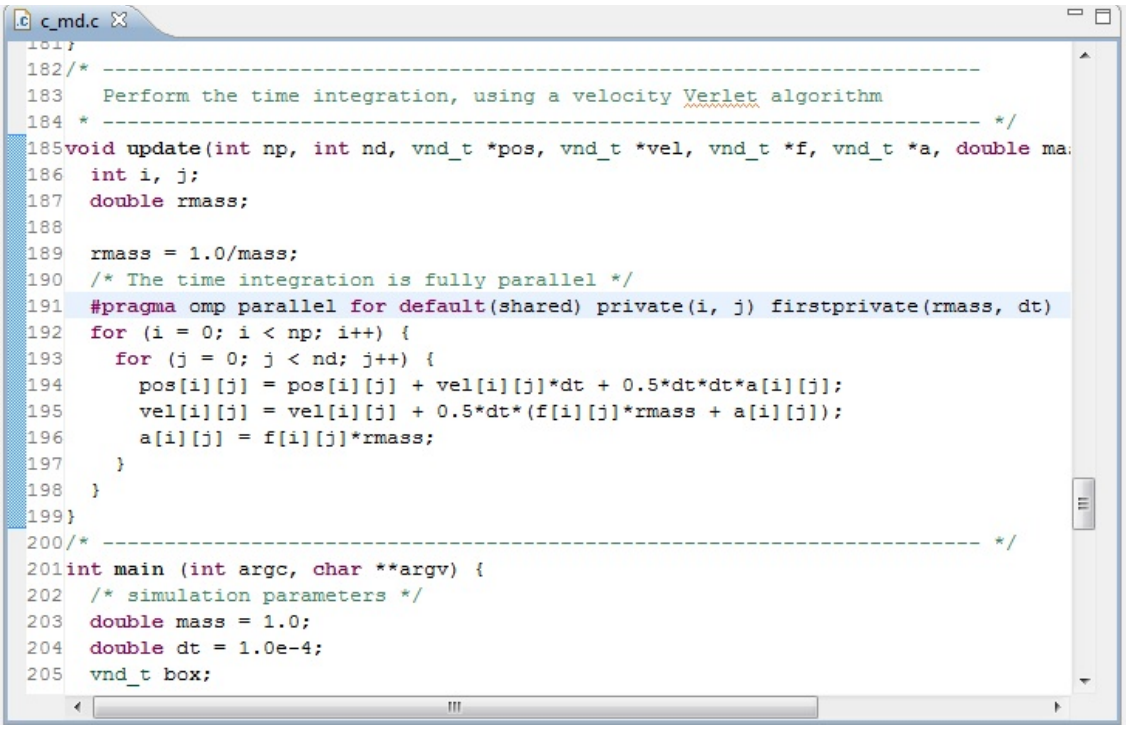

Figure 6.9: Pragma view within the Eclipse editor

with the OmpSCR benchmark. By selecting two or more configurations, that is clicking on two bars in the performance graph, the corresponding source files can be compared. In this case, the source artefact comparing the two different versions of `c_md.c` at the point of the #is shown in Figure 6.10.

The differences between the two source versions, in this simple case, just an OpenMP `#pragma` on line 162 of the left panel, can be highlighted. This allows the user to directly navigate from the performance graph to the source, and view the source changes that may have contributed to the performance differences seen in the graph.

In this example, it is evident that the `FullParallel` versions result in better performance as the two `FullParallel` versions have shorter bars than the versions with partial parallelization. The bar length, the colouring of the selected base bar, and the colour coding of the remaining bars aide in this visual configuration comparison.

Figure 6.10: Navigation with Eclipse

6.4 Case Study: Tracking Changes

In this preliminary case study we investigate the fine-grained evolution of a 3D-SCN-TLM implementation for full optimization on NVIDIA GPU hardware, as presented in the Rossi Masters thesis (Rossi, 2010). As a proof-of-concept, to motivate and evaluate the application of the Rupture model, we focus specifically on kernel optimizations to reduce resource utilization in terms of memory transactions and performance overhead. An overview of the design and implementation decisions as outlined within the Rossi thesis is followed by an analysis across three versions of the 3D-SCN-TLM implementation investigating the changes from the perspective of the Rupture model and assistance from the C_3PO framework.

6.4.1 Optimization Changes to Algorithm Design

Chapter 7 of the Rossi thesis describes the need to make changes to the design and ultimately the implementation of the 3D-SCN-TLM algorithm in order to address issues of global memory resource utilization. Specifically, the early design of the algorithm split the CUDA implementation into separate kernels of core computation to run on the GPUs. While the details of the behavior of this computation will not be discussed, in general terms, the algorithm involves *scattering*, *impulse interchange*, *boundary* calculations and three types of *excitation*.

In a performance analysis, the *scattering* kernel was found to have 80% of its execution consumed by global memory transactions. In order to address this issue, the *scattering* computation was split in two and the *impulse interchange* was subsumed by the *scattering* computation. This change allowed for the iterative and alternating execution of the two phases of *scattering*, decreasing the number of memory transactions. This memory-access optimization is to be performed mainly in phase one of the *scattering*.

The *boundary* and *excitation* computational units required no change in this set of optimizations. Although the *boundary* calculation is described as not requiring change, it is indicated that future considerations of an alternate design could increase execution speed but would in turn consume more memory.

6.4.2 Redesign Forcing Changes to Software Artefacts

In order to investigate the applicability of the Rupture style development model to this type of optimization-focused design change, we consider three snapshots of two of the core CUDA source files as the 3D-SCN-TLM redesign was implemented. This case study provides the opportunity for a two-level investigation of change artefacts at the level of the OPA ontology entities and at the granularity of lines-of-code.

Considering the general description of the design changes as described in Section 6.4.1, we expect the changes to lie within lines of code that could be classified as *task* and *task coordination* under OPA. Specifically, Rossi discusses the need to split up the kernels of core computation and introduce an iterative and alternating execution of this computation. According to OPA, the core computation is classified as a *task* entity and the ordering of the execution of these compute units is classified as *task coordination*.

Table 6.4 provides a summary of the file statistics for this project, showing consistently less than ten source files and the core CUDA files containing less than 700 lines of code (LOC).

Table 6.4: TLM file sizes and function counts for three implementation versions (15, 16 & 17) NOTE: function counts for `Kernel.cu` refer to the number of kernels within the file

Version	Source File Count	File Name	LOC Count	Function Count
15	9	Kernel.cu	454	9
		TLM.cu	196	18
16	8	Kernel.cu	368	9
		TLM.cu	208	19
17	8	Kernel.cu	692	15
		TLM.cu	209	18

Table 6.5 provides a coarse-grained overview of the magnitude of these incremental changes in terms of the number of lines of code, the number of functions added and removed and the number of functions changed. Looking at this data, it becomes evident that substantial change to the kernel functions occurred across the second snapshot of version 16 to 17, while smaller changes occurred across versions 15 and 16, mainly within `TLM.cu`.

In order to understand the details of these changes we must consider these deltas (Δs) at the level of lines of code. Figures 6.11 and 6.12 demonstrate the use of a proprietary tool called WinMerge (WinMerge, 2011) for the visualization of differences across versions. The two tall, rectangular, white boxes on the left conceptually

Table 6.5: Change snapshots across three TLM implementation versions

File	Metric	$\Delta_{15,16}$	$\Delta_{16,17}$
Kernel.cu	LOC change	-86	+287
	Functions added	1	4
	Functions removed	1	0
	Functions changed	2	2
TLM.cu	LOC change	+12	-1
	Functions added	1	0
	Functions removed	0	1
	Functions changed	2	0

represent the two files from the versions under comparison. The colored lines indicate the places where the files differ, with yellow indicating extra code, grey indicating missing code and white showing no change. This perspective is tied to the source code view shown in the right two panels of Figures 6.11 and 6.12 with navigation from the conceptual, file view to the points of change in the source files. The exact differences within a yellow line are indicated with a lighter shade of yellow.

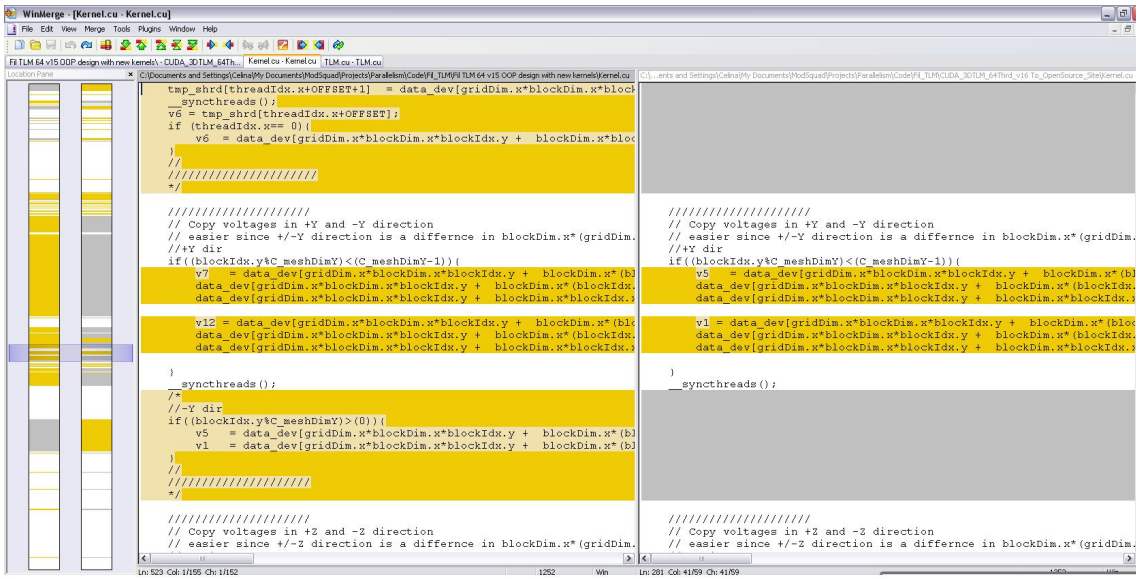

Figure 6.11: Visualisation of Version 15 and 16 differences in 3D-SCN-TLM implementation

In order to understand the details of these changes we delve into the lines of code guided by this visualization of Δsource artefacts. Table 6.6 provides an overview of the details of these changes across the three versions of the `Kernel.cu` and `TLM.cu` files. As this is a proof-of-concept example, this visualization does not currently integrate with the OPA perspective, but as we consider these changes at the granularity of

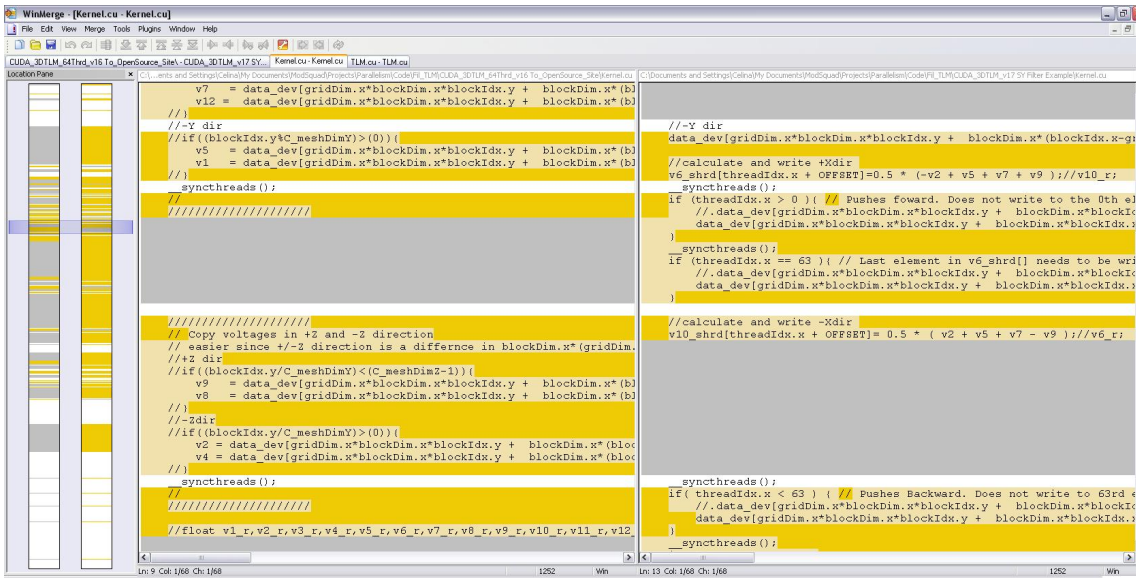

Figure 6.12: Visualisation of differences between version 16 and 17 of 3D-SCN-TLM implementation

lines-of-code, we identify the OPA entity each change would theoretically lie within.

Table 6.6: System snapshot change details

File	$\Delta_{15,16}$	$\Delta_{16,17}$
Kernel.cu	new *boundary kernel* added	*scatter 64 thread 1st pass kernel* added
	second *scatter interchange kernel* removed	*scatter interchange memory test kernel* added
	minor changes to first *scatter interchange kernel*	*scatter interchange 1st pass and 2nd pass kernels* added
	removal of 'save data back to memory' code from *interchange only kernel*	*boundary debug kernel* added
		major changes to *scatter interchange kernel* to introduce 1st pass
		minor changes to *interchange only kernel* to add negative directions
TLM.cu	*interchange only* function added to call *interchange only kernel*	*interchange only* function removed
	minor changes to *boundary execute relay* function to execute second *boundary* kernel added	minor changes to *boundary execute relay* function to experiment with the *boundary debug kernel* and original *boundary kernel*
	minor changes to *iteration relay* function changing placement of synchronization calls	

Column 2 of Table 6.6 outlines the changes made from version 15 to version 16. The addition of a new *boundary* kernel, under inspection was found to be only a slight modification of four lines of code dealing with rounding issues associated with `X` and `Y` coordinate calculations. Inlining of offset calculations of the scatter results to be saved back to global memory were made to the first *scatter interchange* kernel.

The second *scatter interchange* kernel was removed completely. The *interchange only* kernel saw the removal of functionality to transfer data back to memory. The calculation inlining and kernel removal within this change aligns with the *task* entity under an OPA perspective, while the finer details associated with saving data to memory align with the *data coordination* entity of OPA.

Subsequent changes were made in the `TLM.cu` file, with an *interchange only* function added to invoke the new *interchange only* kernel that was added to `kernel.cu`. Minor changes were made to functions invoking the new *boundary* kernel to accommodate the renamed and slightly altered version of this kernel. Finally, minor changes were made to the *iteration relay* function to change the ordering of synchronization calls. The changes to control the use of the altered and added kernel functions across these two versions of `TLM.cu` fall in the classification of *task coordination*. Although the required change to operation ordering is associated with *task coordination*, the change in access to shared data within `Kernel.cu` forces changes to synchronization ordering within `TLM.cu` which aligns with OPA's *synchronization* entity.

The second column of this table overviews the more substantial set of changes made from version 16 to 17. In terms of the kernel code within `Kernel.cu`, a memory test *scatter interchange* kernel and a debugging *boundary* kernel were added across versions. A 64-thread specific, first-pass *scatter* kernel was also added, along with first and second pass *scatter interchange* kernels. Major changes were made to the *scatter interchange* kernel to introduce the first pass and minor changes were made to the *interchange only* kernel. The addition of the two new kernels falls under the *task* entity of OPA while changes to the interchange kernels to account for kernel additions align more with the *task coordination* entity.

Again, subsequent changes were made to the *interchange only* function within the `TLM.cu` file to experiment with the *scatter interchange* memory test and the *boundary* debug kernels. This experimentation was identified through commented out lines-of-code within the source files. This file also witnessed the removal of the *interchange only* function that was added across versions 15 and 16 to invoke the *interchange only* kernel. These changes to `TLM.cu` deal entirely with the invocation and coordination of the kernels and aligns with *task coordination* OPA entity.

6.4.3 Alignment with the Rupture Model

In reflection on the analysis of this change across the three versions of 3D-SCN-TLM, four Rupture change artefacts and causal relationship between them can be identified: $\Delta dynamic$, $\Delta design$, $\Delta source$ and $\Delta profiling$. It was Rossi's dynamic result, demonstrating the overhead of global memory transactions, that initiated this change to the design of the application, causally forcing changes to source and profiling.

The description of design choices in Section 6.4.1 highlighted a focus on changes to be made to the compute kernels. Section 6.4.2 began by speculating that a visualization of changes at the level of the ontology would narrow a developer's view to only *task* and *task coordination* code. The subsequent analysis of the change artefacts in Section 6.4.2 identifies changes spanning both *data coordination* and *synchronization* in addition to those expected within *task* and *task coordination*.

The implications of this result becomes clearer with the finer grained, lines-of-code analysis of the change artefacts. There is a tight correlation between the changes to the *task coordination* code, which alters the algorithm slightly and requires a change to the definition of the actual computation units within the *task* code. Changes to the *task coordination* are coupled with changes to *data coordination* code related to data transfer between local and global memory. This change in access to shared data could have forced the changes in *synchronization* ordering in the `TLM.cu` source file. These dependencies are illustrated in Figure 6.13.

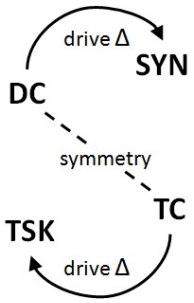

Figure 6.13: OPA entity causal relationships

This result confirms the concern relationships of OPA outlined in Table 6.2 including the dependence of *task* on *task coordination* and *data coordination* and a symmetry between *task coordination* and *data coordination*. Specifically, the changes to the design as described by Rossi, forced changes to source artefacts within *task coordination* and *data coordination* as illustrated in Figure 6.14.

Figure 6.14: Rupture artefact and OPA entity causal relationships

The coupled changes to *task coordination* and *data coordination* is an example of the symmetry between these entities identified in Table 6.2. Further, the changes within *task* were driven by the desire to alternate computation introduced in *task coordination*. This demonstrates the dependence of *task* on *task coordination* identified in the intersection of the *task* row of the *task coordination* column of Table 6.2. Similarly, the intersection of the *synchronization* column with both the *task* and *data coordination* rows indicate a dependency if a shared data structure is in use. An example of this dependency is illustrated by the change to *synchronization* across versions 15 and 16, driven by the changes to *data coordination* that were forced by the change in design. This result demonstrates the need for ontology reasoning to support tools to highlight these dependencies as changes are made within the source.

As described in the case study integrating static and dynamic views in Section 6.3.2, we envision tools that would be able to integrate views of related change artefacts. The results of the above analysis of Rossi's optimization process provides evidence of the correlation between these changes and motivates the need for an integrated tool suite to support the Rupture model. To provide a summary of this evidence, we overview the relationship between Δdesign and Δsource followed by the subsequent correlation to Δdynamic and Δprofiling.

Recalling Rossi's description of the design (Section 6.4.1), it specifically called for: 1) a change to the *scattering* and *impulse interchange* computation splitting the *scattering* into two phases and 2) the subsuming of the *interchange* functionality in order to address memory access issues. Specifically, in the analysis of changes across the three versions outlined in Section 6.4.2, the C_3PO framework could provide a visualization of the consequences of Rossi's design change (Δdesign) within the Δsource artefact. This visualization would highlight the first and second pass *scatter interchange* kernels that were introduced and the *interchange only* function that was removed in Version 17 ($\Delta_{16,17}$). The limited changes made to the *boundary* kernel and no change to the three *excitation* kernels is in keeping with the design decision

in which Rossi indicated no changes to these computational units were necessary. The type of support for visualizing the Δsource artefact shown Figures 6.11 and 6.12 allows a user to skip over the sections that remain the same in a Rupture artefact.

The dynamic nature of the fine-grained optimizations implemented by Rossi tracking a 12-times speed up across incremental changes. This case study demonstrates an iterative feedback, driven by dynamic changes that align with that of the Rupture model. The introduction of a *scattering interchange* memory test kernel and a *boundary* debug kernel are prime examples of the need to profile to capture and understand changes in the dynamic result of an application for both testing and implementation purposes. The change to the source code to monitor subsequent dynamic change would be a tangible change in the Δprofiling artefact according to the Rupture model.

6.5 Summary

In order to address objectives O6, O7 and O8:

- **O6.** Propose a framework that supports the analysis and comparison of key parallel software development artefacts identified.
- **O7.** Evaluate the framework and development process in the context of parallel software artefact examples at the conceptual and implementation level.
- **O8.** Evaluate the framework and development process in the context of existing optimization strategies.

this chapter proposed C_3PO as a framework to support the analysis of critical artefacts proposed in the Rupture model. The application of C_3PO both at the level of conceptual and implementation artefacts combined with its applicability to optimizing approaches serves as an evaluation of the Rupture model.

The contribution of this chapter is the framework for the semi-automated visualization support for the parallel development process. This framework was evaluated in terms of the collection and visualization of change artefacts of the proposed Rupture model. Specifically, this chapter provides an evaluation of the integration of the proposed Rupture development model through tool support leveraging the abstraction provided by OPA. The alignment of the Rupture model with existing parallel optimization strategies confirms the efficacy and suitability of this proposed approach.

This chapter accomplishes the following items in support of the contribution to this thesis:

- Proposed a framework that supports the analysis and comparison key parallel software development artefacts identified.
- Evaluated the framework in the context of parallel software artefact examples at the conceptual and implementation level.
- Evaluated the framework and development model in the context an existing optimization study.

The following chapter provides a summary of the contributions of this thesis and proposes future work to extend these results.

Chapter 7

Conclusion

This chapter provides a summary of the contributions made within each of the chapters presented here and discusses how these contributions support the thesis of this dissertation. This work contributes both a survey of existing work in software practices within the multi-core domain area and proposes viable and vetted support strategies for this problem domain. This dissertation also provides a foundation for future work in support for parallel software development in terms of the identification of the complexities associated utilization of this new age of architectures. A summary of these contributions (Section 7.1) and a plan for future work that builds on these contributions (Section 7.2) follows.

7.1 Summary

Recall, our hypothesis that software development practices and integrated environments can better support parallel development with a model that structures the acquisition and tracking of tangible artefacts key to the parallel development cycle. Moreover, a parallel-specific conceptual model in the form of an ontology can provide a unified view of these artefacts to support both analysis and comparison within and across artefact types. A list of objectives that support these statements was provided in the thesis organization (Chapter 1, Section 1.4) and is duplicated here for convenience.

- **O1.** Determine what is easy/hard about parallel software development.
- **O2.** Discover artefacts key to the parallel software development process.

- **O3.** Discover practical approaches for analysis and comparison of these artefacts.
- **O4.** Discover a conceptual model representative of solutions to parallel problems.
- **O5.** Use this conceptual model to describe and map to multiple artefacts in a uniform way.
- **O6.** Propose a framework that supports the analysis and comparison of key parallel software development artefacts identified.
- **O7.** Evaluate in the context of parallel software artefact examples at the conceptual and implementation level.
- **O8.** Evaluate in the context existing optimization strategies.

Here, we summarize the findings of this research that target each of these objectives, shape the contributions and ultimately provide grounds for the statement of this thesis.

The identification and verification of critical artefacts and the causal relationships within existing parallel software provides a foundation for the proposed Rupture model. Further, the recognition of performance as a key non-functional requirement in the parallel development process guides the identification of feedback loops and demonstrates the need for new workflows guided by the dynamic impact of static changes, forming the basis of the Rupture model.

While the identification of the key artefacts to be tracked is a first step, this thesis goes on to consider practical and systematic approaches for analyzing these artefacts for the purpose of comparison. Beginning with design patterns, the formal pattern language structure is leveraged to establish RIPL, a methodology that isolates and highlights the tradeoffs of a family or group of patterns within a pattern language. This process for comparison is intended to support both pattern selection and composition, but from a wider perspective, demonstrates the ability to compare alternative designs in the Rupture model.

OPA, developed from a combination of real-world scenarios and an initial survey of the parallel programming domain, serves as an abstraction for parallel development that is shown to apply to artefacts at multiple levels of the software development lifecycle. The ability to tie this abstraction to scenarios, source and patterns sets the

stage for the use of the ontology as a conceptual link between the various representations of a solution.

Finally, a framework of tools is proposed to support the analysis and comparison of key parallel software development artefacts, leveraging visualization support and the abstraction provided by OPA. The evaluation of this tool support within existing source code and the alignment of the Rupture model with existing parallel optimization strategies confirms the efficacy and suitability of this proposed approach.

The contributions of this research summarized above are listed explicitly here in terms of the initial objectives of this dissertation.

- **C1.** Documented what is easy/hard about parallel software development.
- **C2.** Discovered artefacts key to the parallel software development process.
- **C3.** Discovered practical approaches for analysis and comparison of these artefacts.
- **C4.** Discovered a conceptual model representative of solutions to parallel problems.
- **C5.** Use this conceptual model to describe and map to multiple artefacts in a uniform way.
- **C6.** Proposed a framework to support the analysis and comparison of key parallel software development artefacts identified.
- **C7.** Evaluated the proposed framework in the context of parallel software artefact examples at conceptual and implementation levels.
- **C8.** Evaluated the proposed framework and the identified artefacts in the context existing optimization strategies.

These contributions resulted from the process of investigating each of the research objectives and consequentially produced further contributions in the form of models, processes and tools. These additional contributions and their relationship to the primary contributions of this research are listed explicitly here.

- **C9.** The Rupture model for parallel software development based on the key artefacts identified and evaluated in contributions **C2** and **C8**.

- **C10.** The RIPL (Relationship Initiated Pattern Language) process for the systematic comparison and analysis of design pattern artefacts based on the approaches identified in contribution **C3**.
- **C11.** The OPA (Ontology for Parallel Applications) ontology that applies to parallel applications, based on the conceptual model discovered and evaluated in contributions **C4** and **C5**.
- **C12.** The C_3PO (Communication, Computation and Coordination for Parallel Objectives) framework for the automation of parallel software artefact analysis and comparison proposed and evaluated in contributions **C6** and **C7**.

7.2 Future Work

The work supporting this thesis has made several contributions to the community of researchers studying software engineering elements of parallel development. Our study of code bases and conceptual models identified several core concepts central to parallel computing in a language/platform agnostic context (Gibbs *et al.*, 2009). Our follow up work on this conceptual model proposed entities and relationships in an ontology that bridged the gap between implementation and design tools to support program comprehension and reasoning (Gibbs and Coady, 2010a).

Our work with design patterns proposed a systematic process for pattern selection based on forces explicitly identified within a pattern language (Gibbs and Coady, 2010b, 2009). A case study applying the proposed methodology to two existing pervasive pattern languages revealed the ability to represent pattern relationships in a structured, systematic way. It is still an open question as to whether or not these approaches will scale to include the parallel design patterns being constructed by other stake-holders in the community, such as the Berkeley ParLab and Microsoft. We continue to research alternative ways to represent the information offered up through design patterns to the mainstream programmer (Long *et al.*, 2011).

Parallel specific studies revealed the number of artefacts contributing to fundamental dynamic properties in a parallel application tend to be much higher and impact the complexity of the development lifecycle of these applications (Gibbs and Coady, 2010c). Furthermore, the systematic use of existing practices to avoid complexity in sequential systems, such as object-oriented strategies, makes clean parallelization of scientific applications challenging (Chester *et al.*, 2010).

7.2.1 Research Objectives

This proposal for future work aims to build on the foundation for software practices within the multi-core domain established in this thesis. The goal is to leverage a combination of abstraction and tools to collect, monitor and analyse results of the critical indicators of the parallel development process as identified in the Rupture model proposed in this thesis. Specifically, these next steps look to establish concrete ways to facilitate a parallel development model that will scale to real-world applications. In terms of scale, this project is not only concerned with application size but mainly with problem size. That is, we draw motivation from scientific computing to explore the use of the Rupture model as a practical means of delivering better quality of service to large-scale scientific applications that deal with typically massive data sets exceeding current commodity standards. This proposed research project is structured in terms of three supporting objectives:

1. Formalization of our proposed parallel ontology to allow for use of existing analysis tools.

2. Clear and uniform representation of design patterns to allow for the identification of low-level architecture specific tradeoffs.

3. Application of tools and abstraction to real-world scientific applications to establish a concrete representation of the Rupture model focused on dynamic monitoring.

Our proof-of-concept, parallel ontology (OPA), developed to date has been shown to apply at multiple levels of abstraction within the software development lifecycle including design patterns and source code. Our long-term goal to extend this work is to formalize OPA for integration with existing ontology tools to semi-automate analysis of parallel source code. The short-term goal to support this objective is a deeper investigation of the relationships between the components that make up a parallel application that correspond directly to the entities of the OPA ontology. These entity and relationship definitions will serve to map to a parallel code base through existing tools and not only provide a navigable representation of the source but also highlight relationship violations.

We also want to focus on the applicability of current design patterns on problems relative to scientific computing. Our long term goal looks to provide systematic and

accessible use of design patterns by scientific programmers in both individual use and composition. In support of this long-terms goal, we begin by researching uniform ways to represent a pathway of use through a given design pattern.

Our goal is not only to develop tools and abstraction as described in the above goals but to further develop a process and model to support the development of parallel software. Based on the fundamental importance of dynamic feedback into static software artefacts established in my thesis work, our long-term goal is to develop a concrete manifestation of the proposed Rupture model. Short-term goals to support this objective will investigate the use of tools for visualization of static software artefacts with respect to dynamic results at multiple levels of abstraction to both flesh out and further evaluate the Rupture model.

7.2.2 Pertinent Literature

Languages for developing formal ontologies include Protege (Protegé, 2010), Knowledge Interchange Format (KIF) (Genesereth *et al.*, 1992), and Web Ontology Language (OWL) (OWL, 2010). Specifically, the Protege platform supports the modeling of ontologies through the construction of a domain model through the formal definition of concepts or entities and the relationships that are important in that particular domain. While these tools are currently applied to taxonomies, classifications, database schemas and more we believe our work to demonstrate the use of an ontology within the parallel domain sets the ground work to leverage diagnostic tools, such as Chimaera (McGuiness *et al.*, 2000) that check for logical correctness, identifying common errors in common practices.

This work is fueled by others attempting to make the use of patterns specifically for composition purposes. Specifically, Robinson and Johnson propose a meta-design pattern, "Three Layer Cake", for dealing with the complexity of shared-memory programming (Robison and Johnson, 2010). This proposed approach uses Message Passing, Fork-Join and SIMD parallelism in a hierarchy to allow for a high level of parallelism. We plan to leverage the essential set of design patterns for parallel software design currently under construction as a Design Pattern Language from Berkeley (Keutzer and Mattson, 2010).

Prior research with tools like BEAGLE (Tu and Godfrey, 2002) has shown the need of source-based analysis. However, some software concerns like fine grained source level configuration may be highly scattered across many files, and concerns may not

be easily conceptualized by viewing source code alone. Consequently, source analysis tools may opt for higher level views of a software system that take into account complex structures, dependencies and flows that are not immediately discerned from just viewing the source code.

Dealing with source at the level of assembly further complicates these challenges with the presence of heterogeneities between data structures resulting in loss of or falsification of information. Euzenat and Shvaiko (Euzenat and Shvaiko, 2007) provide a general introduction to the subject for inter-operability using ontologies, and Dorion et al (Dorion *et al.*, 2010) provide a detailed characterization of the possible heterogeneity types, leading to a measure of how much information is lost. Defining modeling meta-data to leverage tooling functionality and UI with higher level views of the underlying software system is used in existing frameworks like IBMs RAD (Rodriguez *et al.*, 2005), whereas other tools work on intermediate object representations of a code base, like Eisenbarths C concept analysis tool (Eisenbarth *et al.*, 2001).

7.2.3 Methods and Proposed Approach

Specifically, this approach will begin by considering patterns that apply directly to the scientific community, for example the linear algebra problems relevant to image processing associated with telescope technology (Herriot *et al.*, 2010). This research in design pattern use is tightly tied to and driven by implementation level studies which also form the basis for research in software development models. This approach will start with fundamental smaller, sub-problems, and scale up to larger-scale scientific applications. We hope to further this work to continue to explore the main goals of parallel pattern designers: education of parallel programming, communication between parallel programmers and design of parallel programs.

The method to address the goals of this project will be case study driven with the application of design patterns to scientific computing problems. Leveraging core computational problems within scientific computing as the foundation for this work, we propose to analyze the concrete examples of optimization to experiment with meaningful ways of representing correlated static and dynamic results. We will continue along the lines of a framework based approach, investigating it in terms of the extensibility of its application to software artefacts including design patterns and source code written in both high-level languages and low-level assembly generated from a de-compilation process.

7.2.4 Significance of this Work

The premise of this work is that the challenge of developing parallel software that is accessible to large-scale scientific applications running on clusters of nodes with multi-core processors can be met by carefully architecting software. Specifically, I plan to coordinate emerging efforts of a group of graduate students at UVic focusing on this work. The goal will be to research systematic methodologies augmented by a parallel specific software development lifecycle and guided by a design pattern language. To this end, the plan is to experiment with real-world scientific problems and the optimization of their parallel solutions. While doing so, we aim to identify where patterns help and hinder design decisions, and develop tools to support an optimization driven software development process.

Bibliography

Adams, B. (2008). *Co-evolution fo Source Code and the Build System: Impact of the Introduction of AOSD in Legacy Systems*. Ph.D. thesis, Ghent University, Belgium.

Agerbo, E. and Cornils, A. (1998). How to preserve the benefits of design patterns. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'98)*, pages 134–143.

Alexander, C. (1979). *The Timeless Way of Building*. Oxford University Press, New York.

Alexander, C., Ishikawa, S., and Silverstein, M. (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *American Federation of Information Processing Society (AFIPS)*, pages 483–486, Anaheim, California, USA.

Andrade, H. (2009). Discrete event. In *Proceedings of Workshop on Parallel Programming Patterns (ParaPLOP'09)*.

Andrews, T., Qadeer, S., Rajamani, S. K., Rehof, J., and Xie, Y. (2004). Zing: A model checker for concurrent software. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'04)*, pages 484–487, Boston, MA, USA. Springer.

Android (2008). Android – An Open Handset Alliance Project. http://code.google.com/android/what-is-android.html. Date accessed: Nov. 2008.

Ansaloni, D., Binder, W., Villazón, A., and Moret, P. (2010). Parallel dynamic analysis on multicores with aspect-oriented programming. In *Proceedings of the*

International Conference on Aspect-Oriented Software Development (AOSD'09), pages 1–12, Rennes and Saint-Malo, France. ACM.

Apache (2007a). Apache Harmony – open source Java platform. http://harmony.apache.org/. Date accessed: Sep. 2007.

Apache (2007b). Apache Portable Runtime Project. http://apr.apache.org/. Date accessed: Sep. 2007.

Appavoo, J., Uhlig, V., and Waterland, A. (2008). Building a global-scale computer. *SIGOPS Operating System Review*, **42**(1), 77–84.

Apple (2008). OpenCL. http://www.apple.com/server/macosx/snowleopard/. Date accessed: Mar. 2009.

Armstrong, J. (2007). A history of Erlang. In *Proceedings of the third ACM SIGPLAN Conference on the History of Programming Languages (HOPL III)*, pages 6–1–6–26, New York, NY, USA. ACM.

Arpaci-Dusseau, R. H., Anderson, E., Treuhaft, N., Culler, D. E., Hellerstein, J. M., Patterson, D., and Yelick, K. (1999). Cluster I/O with River: making the fast case common. In *Proceedings of the Workshop on I/O in Parallel and Distributed Systems (IOPADS'99)*, pages 10–22, New York, NY, USA. ACM.

Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. (2006). The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

Backus, J. (1978). Can programming be liberated from the von Neumann Style? A functional style and its algebra of programs. *Communications of the ACM*, **21**(8), 613–641.

Bailey, D. H., Barscz, E., Barton, J., Browning., D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., and Weeratunga, S. (1994). The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center), address = Moffett Field, CA.

Baniassad, E. L., Murphy, G. C., and Schwanninger, C. (2003). Design pattern rationale graphs: Linking design to source. In *Proceedings of the International Conference on Software Engineering (ICSE '03)*, pages 352–362, Los Alamitos, CA, USA. IEEE Computer Society.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001). Manifesto for Agile Software Development. http://www.agilemanifesto.org/. Date accessed: Nov. 2010.

Behren, R. V., Condit, J., Zhou, F., Necula, G. C., and Brewer, E. (2003). Capriccio: Scalable threads for internet services. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'03)*, pages 268–281. ACM.

Berkeley Parallel Computing Lab (2010). A Pattern Language for Parallel Programming version 1.0. http://parlab.eecs.berkeley.edu/wiki/patterns/pattern1_0.

Bernstein, P. A. and Melnik, S. (2007). Model management 2.0: Manipulating richer mappings. In *Proceedings of the International Conference on Management of Data (SIGMOD '07)*, pages 1–12, New York, NY, USA. ACM SIGMOD.

Bhansali, S., Chen, W.-K., de Jong, S., Edwards, A., Murray, R., Drinić, M., Mihočka, D., and Chau, J. (2006). Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the International Conference on Virtual Execution Environments (VEE'06)*, pages 154–163, New York, NY, USA. ACM.

Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. (2006). The DaCapo Benchmarks: Java benchmarking development and analysis. In *Proceedings of the Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*, pages 169–190, New York, NY, USA. ACM SIGPLAN.

Boehm, B. W. (1988). A spiral model of software development and enhancement. *Computer*, **21**, 61–72.

Bradley, T. (2009). Intel 48-Core "Single-Chip Cloud Computer" improves power efficiency. *PC World*.

Bragdon, A., Zeleznik, R., Reiss, S. P., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., and LaViola, J. J. (2010a). Code Bubbles: A working set-based interface for code understanding and maintenance. In *Proceedings of the 28th International Conference on Human Factors in Computing Systems (CHI'10)*, pages 2503–2512, New York, NY, USA. ACM.

Bragdon, A., Reiss, S. P., Zeleznik, R., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., and LaViola, J. J. (2010b). Code Bubbles: rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE'10)*, pages 455–464, New York, NY, USA. ACM/IEEE.

Brooks, F. P. (1995). *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley Professional.

Brooks, Jr., F. P. (1975). *The Mythical Man-Month*. Addison-Wesley.

Brooks, Jr., F. P. (1987). No silver bullet: Essence and accidents of software engineering. *Computer*, **20**, 10–19.

Brownsword, A. (2011). Invited Talk: Software architecture in game development. In *Proceedings of Software Engineering and Management Conference: Software Meets Business (OOP)*, Munich, Germany.

Business Objects (2007). The Open Quark Framework for Java and the CAL Language. http://labs.businessobjects.com/cal/.

Chen, N., Karmani, R. K., Shali, A., Su, B.-Y., and Johnson, R. (2009). Collective communication patterns. In *Proceedings of the Workshop on Parallel Programming Patterns (ParaPLOP'09)*.

Chester, S., Gibbs, C., Rossi, F., Brownsword, A., So, P., Gulliver, A., and Coady, Y. (2010). Insulating the scientific programmer from perilous parallel architecture. In *Proceedings of the Workshop of Parallel Object-Oriented Programming for Scientific Computing (POOSC'10), held in conjunction with Systems, Programming Languages, Applications: Software for Humanity (SPLASH)*, Reno, NV, USA.

Chong, J. (2009a). Speculation. In *Proceedings of the Workshop on Parallel Programming Patterns (ParaPLOP'09)*.

Chong, J. (2009b). Task parallelism. In *Proceedings of the Workshop on Parallel Programming Patterns (ParaPLOP'09)*.

Chow, J., Garfinkel, T., and Chen, P. M. (2008). Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA. USENIX Association.

Cooley, J. W. and Tukey, J. W. (1965). An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, **19**(90), 297–301.

Dahl, O.-J. and Nygaard, K. (1966). SIMULA - an ALGOL-based simulation language. *Communications of the ACM*, **9**(9), 671–678.

Dahl, O.-J. and Nygaard, K. (2003). SIMULA. In *Encyclopedia of Computer Science*, pages 1576–1578. John Wiley and Sons Ltd., Chichester, UK.

Darema, F. (2001). The SPMD model : Past, present and future. In *PVM/MPI: Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2131 of *Lecture Notes in Computer Science*, page 1, Santorini/Thera, Greece. Springer.

Davis, S. and Kiczales, G. (2010). Registration-based language abstractions. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10)*, pages 754–773, New York, NY, USA. ACM.

Dean, J. and Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, pages 137–150, San Francisco, CA, USA. USENIX.

Deugo, D., Weiss, M., and Kendall, E. (2001). Reusable patterns for agent coordination. In *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 347–368. Springer.

Dijkstra, E. W. (1965). The multiprogramming system for the EL X8 THE. circulated privately.

Dorion, E., Grenier, D., Brodeur, J., and O'Brien, D. (2010). A Taxonomy of Heterogeneity Types for Ontological Alignment. In preparation.

Dorta, A. J., Rodriguez, C., and de Sande, F. (2005). The OpenMP source code repository. In *Proceedings of the Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP '05)*, pages 244–250.

Duffy, J. and Essey, E. (2007). Running Queries On Multi-Core Processors. http://msdn.microsoft.com/en-us/magazine/cc163329.aspx. Date accessed: Jun. 2009.

Dunlap, G. W., King, S. T., Cinar, S., Basrai, M. A., and Chen, P. M. (2002). ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI'02)*, volume 36, pages 211–224.

Eclipse (2010a). The CDT Project. http://www.eclipse.org/cdt. Date accessed: Nov. 2010.

Eclipse (2010b). The Eclipse Framework. http://www.eclipse.org/. Date accessed: Nov. 2010.

Eichenberger, R. E., Wu, P., Chen, T., Oden, P. H., Prener, D. A., Shepherd, J. C., So, B., Sura, Z., Wang, A., Zhang, T., Zhao, P., and Gschwind, M. (2005). Optimizing compiler for the cell processor. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 161–172. IEEE Computer Society.

Eisenbarth, T., Koschke, R., and Simon, D. (2001). Derivation of feature component maps by means of concept analysis. In *In Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pages 176–179. Society Press.

Engler, D. and Ashcraft, K. (2003). RacerX: Effective, static detection of race conditions and deadlocks. *SIGOPS Operating Systems Review*, **37**(5), 237–252.

Ericsson Computer Science Laboratory (2010). Ericsson - A world of communication. http://www.ericsson.com/.

Ernst, M. D., Badros, G. J., and Notkin, D. (2002). An empirical analysis of C preprocessor use. *IEEE Transactions Software Engineering*, **28**(12), 1146–1170.

Euzenat, J. and Shvaiko, P. (2007). *Ontology Matching*. Springer-Verlag, Heidelberg (DE).

Falconer, S. (2009). *Cognitive support for semi-automatic ontology mapping,*. Ph.D. thesis, Department of Computer Science, University of Victoria.

Fenlason, J. and Stallman, R. (1998). GNU gprof. Technical report, Free Software Foundation.

Flanagan, C. and Qadeer, S. (2003). Types for atomicity. In *Proceedings of the International Workshop on Types in Languages Design and Implementation (TLDI'03)*, pages 1–12, New York, NY, USA. ACM SIGPLAN.

Foster, I. (1995). *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing, Boston, MA, USA.

Fowler, M. (2010). *Domain Specific Languages*. Addison-Wesley Professional, 1st edition.

Frigo, M. and Johnson, S. G. (2005). The design and implementation of FFTW3. *Proceedings of the IEEE*, **93**(2), 216–231. special issue on "Program Generation, Optimization, and Platform Adaptation".

Furlinger, K., Gerndt, M., and Munchen, T. U. (2005). ompP: A profiling tool for OpenMP. In *Proceedings of the International Workshop on OpenMP (IWOMP '05)*, Eugene, Oregon, USA.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Professional.

Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. (2003). The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the conference on Programming Language Design and Implementation (PLDI'03)*, pages 1–11, New York, NY, USA. ACM SIGPLAN.

Gay, D., Levis, P., and Culler, D. (2007). Software design patterns for TinyOS. *Transactions on Embedded Computing Systems*, **6**.

Genesereth, M., Fikes, R. E., Brachman, R., Gruber, T., Hayes, P., Letsinger, R., Lifschitz, V., Macgregor, R., Mccarthy, J., Norvig, P., and Patil, R. (1992). knowledge interchange format version 3.0 reference manual.

George, D. D., Mozdzynski, G., and Salmond, D. (1999). Implementation and performance of OpenMP in ECMWF's IFS code. In *Proceedings of 5th European SGI/Cray MPP Workshop*, Bologna, Italy.

Gibbs, C. and Coady, Y. (2009). Joining forces: A RIPPL effect? Aconstraint-oriented perspective on a pervasive pattern language. volume 0, pages 214–219, Los Alamitos, CA, USA. IEEE Computer Society.

Gibbs, C. and Coady, Y. (2010a). Concurrency conundrums - An ontological solution? In *Proceedings of International Conference on Knowledge Engineering and Ontology Development (KEOD'10)*, Valencia, Spain.

Gibbs, C. and Coady, Y. (2010b). May the Force(s) be with you: A systematic approach to pattern selection. In *Proceedings of the Workshop on Parallel Programming Patterns (ParaPLOP'10)*.

Gibbs, C. and Coady, Y. (2010c). Parallelization and the application programmer: Random self-oscillation or Old Faithful? In *Proceedings of Concurrency for the Application Programmer (CAP'10) held at Systems, Programming Languages, Applications: Software for Humanity (SPLASH'10)*, Reno, NV, USA.

Gibbs, C., Baldwin, J., Singh, N., D'Hondt, M., and Coady, Y. (2008). Living with the Law: Can automation give us Moore with less? In *International Conference on Automated Software Engineering (ASE'08)*, volume 0, pages 395–398. IEEE Computer Society.

Gibbs, C., Gunion, K., and Coady, Y. (2009). On the codification of coordination: An ontological tool. In *Proceedings of the Workshop on Parallel Programming Patterns (ParaPLOP'09)*.

GNU Project (2010a). GCCL: The GNU Compiler Collection. http://gcc.gnu.org/. Date accessed: Nov. 2010.

GNU Project (2010b). GDB: The GNU Project Debugger. http://www.gnu.org/software/gdb/. Date accessed: Nov. 2010.

Goldsmith, S. F., O'Callahan, R., and Aiken, A. (2005). Relational queries over program traces. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 385–402, New York, NY, USA. ACM.

Goldstine, H. and Goldstine, A. (1996). The Electronic Numerical Integrator and Computer (ENIAC). *IEEE Annals of the History of Computing*, **18**(1), 10–16.

Google (2009). The Go programming language. http://golang.org/. Date accessed: Mar. 2009.

Google Groups (2010). Go language nuts. http://groups.google.com/group/golang-nuts. Date accessed: Mar. 2009.

Gordon, M. I., Thies, W., Karczmarek, M., Lin, J., Meli, A. S., Lamb, A. A., Leger, C., Wong, J., Hoffmann, H., Maze, D., and Amarasinghe, S. (2002). A stream compiler for communication-exposed architectures. *ACM SIGOPS Operating Systems Review*, **36**(5), 291–303.

Gruber, T. (2009). Ontology. In *Encyclopedia of Database Systems*, pages 1963–1965. Springer.

Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge Acquisition*, **5**, 199–220.

Gruber, T. R. (1995). Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies – Special Issue: the role of formal ontology in the information technology*, **43**, 907–928.

Gruener, W. (2006). Intel aims for 32 cores by 2010. http://www.tgdaily.com/content/view/27460/135/.

Gunion, K. (2009). *FUNdamentals of CS: Designing and Evaluating Computer Science Activities for Kids*. Master's thesis, Department of Computer Science, University of Victoria.

Gürsoy, A. and Kale, L. V. (2004). Performance and modularity benefits of message-driven execution. *Journal of Parallel Distributed Computing*, **64**(4), 461–480.

Hampton, J. A. (1979). Polymorphous concepts in semantic memory. *Journal of verbal learning and verbal behavior*, **18**, 441–461.

Harris, M. (2005). Mapping computational concepts to gpus. In *Proceedings of the International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'05)*, New York, NY, USA. ACM.

Harris, T. and Fraser, K. (2003). Language support for lightweight transactions. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'03)*, pages 388–402. ACM SIGPLAN.

Havelund, K., Majumdar, R., and Palsberg, J., editors (2008). *Proceedings of the International SPIN Workshop on Model Checking of Software*, volume 5156 of *Lecture Notes in Computer Science*, Los Angeles, CA, USA. Springer.

Heisenberg, W. (1927). über den anschaulichen Inhalt der quantentheoretischen Kinematik und Mechanik. *Zeitschrift für Physik*, **43**(3-4), 172–198.

Herlihy, M. and Moss, J. E. B. (1993). Transactional memory: Architectural support for lock-free data structures. In *Proceedings of International Symposium on Computer Architecture*, pages 289–300, San Diego, CA, USA. IEEE Computer Society Press.

Herriot, G., Andersen, D., Atwood, J., Boyer, C., Beauvillier, A., Byrnes, P., Conan, R., Ellerbroek, B., Fitzsimmons, J., Gilles, L., Hickson, P., Hill, A., Jackson, K., Lardière, O., Pazder, J., Pfrommer, T., Reshetov, V., Roberts, S., Véran, J., Wang, L., and Wevers, I. (2010). NFIRAOS: TMT's facility adaptive optics system. In *Proceedings of Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 7736. SPIE.

Hillside Group (2010). Home of the design patterns library and host of the PLoP conferences. http://hillside.net/.

Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, **21**(8), 666–677.

Holzmann, G. J., Joshi, R., and Groce, A. (2008). Swarm verification. In *Proceedings of the International Conference on Automated Software Engineering (ASE '08)*, pages 1–6. IEEE.

Holzmann, G. J., Joshi, R., and Groce, A. (2010). Swarm verification techniques. *Transactions on Software Engineering*, **99**(PrePrints).

Hovey, G. J., Conan, R., Gamache, F., Herriot, G., Ljusic, Z., Quinn, D., Smith, M., Veran, J. P., and Zhang, H. (2010). An FPGA based computing platform for Adaptive Optics control. In *Proceedings of Conference on Adaptive Optics for Extremely Large Telescopes (AO4ELT '10)*. EDP Sciences.

Hu, Y., Merlo, E., Dagenais, M., and Lagüe, B. (2000). C/C++ conditional compilation analysis using symbolic execution. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 196, Washington, DC, USA. IEEE Computer Society.

Hutchinson, N. C. (1987). *EMERALD: An object-based language for distributed programming*. Ph.D. thesis, University of Washington, Seattle, WA, USA.

IBM (1988). *Rational Unified Process: Best Practices for Software Development Teams*. Rational Software Corporation, Cupertino, CA.

IBM (2008). The Cell project at IBM Research. http://www.research.ibm.com/cell/.

Institute of Electrical and Electronics Engineers (2004). Standard for information technology - portable operating system interface (POSIX). *IEEE Std 1003.1, 2004 Edition. The Open Group Technical Standard. Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. Shell and Utilities*.

Intel Corporation (2005). Excerpts from a conversation with Gordon Moore: Moore's Law. ftp://download.iintel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_converation_with_Gordon_Moore.pdf.

Intel Corporation (2008a). Ct: C for throughput computing. http://techresearch.intel.com/ articles/Tera-Scale/1514.htm. Date accessed: Mar. 2009.

Intel Corporation (2008b). VTune Performance Analyzer. http://www.intel.com/cd/software/products/asmo-na/eng/239144.htm. Date accessed: Mar. 2008.

Isard, M. and Birrell, A. (2007). Automatic Mutual Exclusion. In *Proceedings of the 11th USENIX workshop on Hot topics in Operating Systems (HOTOS'07)*, pages 1–6, Berkeley, CA, USA. USENIX.

Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. (2007). Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the European Conference on Computer Systems (EuroSys'10)*, Lisbon, Portugal. Microsoft Research, Silicon Valley, ACM.

Jacobson, I. (1999). *The Unified Software Development Process*. Addison-Wesley, Boston, MA, USA.

Johnson, R. (2009). Using a pattern language to design a system. http://www.cincomsmalltalk.com/userblogs/ralph.

Johnson, R., Keutzer, K., and Mattson, T., editors (2009). *Workshop on Parallel Programming Patterns (ParaPLOP)*.

Johnson, R., Keutzer, K., and Mattson, T., editors (2010). *Workshop on Parallel Programming Patterns (ParaPLOP)*.

Kale, V. (2010). Scalable sorting pattern. In *Proceedings of the Workshop on Parallel Programming Patterns (ParaPLOP'10)*.

Kernighan, B. W. and Ritchie, D. (1978). *The C Programming Language*. Prentice-Hall.

Keutzer, K. (2010). Architecting parallel software with patterns. http://embedded.eecs.berkeley.edu/eecsx44/lectures/Kurt-Lecture.pdf.

Keutzer, K. and Mattson, T. (2010). A design pattern language for engineering (parallel) software. *Intel Technology Journal: Addressing the Challenges of Terascale Computing*, **13**(4).

Kiemele, L. (2009). A Comparison of Parallel Programming Technologies with PThreads, OpenMP and OpenCL. Technical report, University of Victoria, Victoria, BC, CAN.

King, S. T., Dunlap, G. W., and Chen, P. M. (2005). Debugging operating systems with Time-Traveling virtual machines. In *Proceedings of USENIX Annual Technical Conference, General Track*, pages 1–15.

Kirk, D. B. and Hwu, W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 1st edition.

Krawezik, G. and Cappello, F. (2006). Performance comparison of MPI and OpenMP on shared memory multiprocessors. *Concurrency and Computation: Practice and Experience*, **18**(1), 29–61.

Kruchten, P. (2003). *The Rational Unified Process: An Introduction*. Addison-Wesley, Boston, MA, USA, 3rd edition.

Kuhn, B., Petersen, P., and O'Toole, E. (2000). OpenMP versus threading in C/C++. *Concurrency – Practice and Experience*, **12**(12), 1165–1176.

Lauer, H. C. and Needham, R. M. (1979). On the duality of operating system structures. *SIGOPS Operating Systems Review*, **13**(2), 3–19.

Lavender, G. R. and Schmidt, D. C. (1995). Active Object: an Object Behavioral Pattern for Concurrent Programming. In *Proceedings of Pattern Languages of Programs*.

Lefebvre, G., Cully, B., Feeley, M. J., Hutchinson, N. C., and Warfield, A. (2009). Tralfamadore: Unifying source code and execution experience. In *Proceedings of the EuroSys Conference*, pages 199–204, Nuremberg, Germany. ACM.

Leijen, D. and Hall, J. (2007). Optimize managed code for multi-core machines. http://msdn.microsoft.com/en-us/magazine/cc163340.aspx.

Lewis, J. J. (2010). Grace Hopper quotes. http://womenshistory.about.com/od/quotes/a/grace_hopper.htm. Date accessed: Dec. 2010.

Lewis, T. and El-Rewini, H. (1993). Parallax: A tool for parallel program scheduling. *IEEE Parallel Distributed Technology*, **1**(2), 62–72.

Liskov, B. (1988). Distributed programming in Argus. *Communications of the ACM*, **31**(3), 300–312.

Long, D. K., Kiemele, L., Gibbs, C., Brownsword, A., and Coady, Y. (2011). Mind the Gap! Bridging the dichotomy of design and implementation. In *Proceedings of the Workshop on Software Engineering for Computational Science and Engineering (SECSE'11) held at the International Conference on Software Engineering (ICSE'11)*.

Lu, S., Park, S., Hu, C., Ma, X., Jiang, W., Li, Z., Popa, R. A., and Zhou, Y. (2007). MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*, pages 103–116, New York, NY, USA. ACM.

Martin, M., Livshits, B., and Lam, M. S. (2005). Finding application errors and security flaws using pql: a program query language. In *Proceedings of the ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 365–383, New York, NY, USA. ACM.

Mattson, T., Sanders, B., and Massingill, B. (2004). *Patterns for parallel programming*. Addison-Wesley Professional, first edition.

McCarthy, J. (1978). History of LISP. *SIGPLAN Notices*, **13**, 217–223.

McCarthy, J. (1980). Circumscription – A form of non-monotonic reasoning. *Artificial Intelligence*, **13**(1-2), 27–39.

McGuiness, D. L., Fikes, R., Rice, J., and Wilder, S. (2000). An environment for merging and testing large ontologies. In *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning*, pages 483–493, Colorado,USA.

Microsoft (2008). Microsoft Visual Studio. http://msdn.microsoft.com/en-us/vstudio/products/default.aspx. Date accessed: Mar. 2008.

Milner, R. (1989). *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Monteyne, M. (2008). RapidMind multi-core development platform. *RapidMind White Paper*.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, **38**(8), 114–117.

Murphy, G. L. (2002). *The Big Book of Concepts (Bradford Books)*. MIT Press.

Murthy, P. (2008). Parallel computing with X10. In *Proceedings of the International Workshop on Multicore Software Engineering (IWMSE'08)*, pages 5–6, New York, NY, USA. ACM.

NASA Advanced Supercomputing Division (1999). NASA Advanced Supercomputing (NAS) Parallel Benchmarks (NPBs). http://www.nas.nasa.gov/Resources/Software/npb.html. Date accessed: Mar. 2008.

National Research Council Canada (NRCC) (2010). Adaptive Optics (AO) programs. http://www.nrc-cnrc.gc.ca/eng/programs/hia/adaptive-optics.html.

Neches, R., Fikes, R., Finin, T., Gruber, T., Patil, R., Senator, T., and Swartout, W. R. (1991). Enabling technology for knowledge sharing. *AI Magazine*, **12**, 36–56.

Nethercote, N. and Seward, J. (2007). Valgrind: Aframework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'07)*, pages 89–100, New York, NY, USA. ACM.

Netzer, R. H. B. and Miller, B. P. (1992). What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, **1**(1), 74–88.

Noy, N. and McGuiness, D. (2001). Ontology development 101: A guide to creating your first ontology. Technical Report SMI-2001-0880, Stanford Medical Informatics, Stanford University, USA.

NVIDIA Corporation (2008). The NVIDIA Tesla Supercomputer. http://www.nvidia.com /object/personal_supercomputing.html.

NVIDIA Corporation (2009). CUDA C Best Practices Guide. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA. Date accessed: Mar. 2009.

NVIDIA Corporation (2010). CUDA Zone. www.nvidia.com/object/cuda_home.html. Date accessed: Mar. 2009.

Nygaard, K. (2001). OO-pedagogical approach. http://www.exciton.cs.rice.edu/comp410/Project/OO-PedagogicalApproach.ppt. Date accessed: Dec. 2010.

Nygaard, K. (2002). COOL (Comprehensive Object-Oriented Learning). In *Proceedings of the 7th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, page 218, Aarhus, Denmark. ACM.

O'Callahan, R. (2008). Chronomancer: C/C++ trace-based debugger based on Chronicle and Eclipse. http://code.google.com/p/chronomancer. Date accessed: Dec. 2010.

O'Hanlon, C. (2006). A conversation with John Hennessy and David Patterson. *Queue*, **4**, 14–22.

Ompf (2010). AMD Fusion – Potentially interesting platform for ray tracing. http://ompf.org/forum/viewtopic.php?t=1923&p=21374. Date accessed: Jan. 2011.

OpenMP Architecture Review Board (2005). OpenMP application program interface. Technical Report 2.5, OpenMP Architecture Review Board.

OProfile (2010). OProfile – A System Profiler for Linux. http://oprofile.sourceforge.net/news/. Date accessed: Nov. 2010.

Ousterhout, J. K. (1996). Invited Talk: Why threads are a bad idea (for most purposes). In *USENIX Technical Conference*.

OWL (2010). OWL (Web Ontology Language). http://infomesh.net/2001/swintro/. Date accessed: Nov. 2009.

Palmer, S. B. (2001). The semantic web: An introduction. http://infomesh.net/2001/swintro/. Date accessed: May 2011.

Pancake, C. M. and Bergmark, D. (1990). Do parallel languages respond to the needs of scientific programmers? *Computer*, **23**, 13–23.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, **15**(12), 1053–1058.

PCI SISIG (2010). PCI Express 3.0 frequently asked questions. http://www.pcisig.com/news_room/faqs/pcie3.0_faq/. Date accessed: Jan. 2011.

Pialorsi, P. and Russo, M. (2007). *Introducing Microsoft®Linq*. Microsoft Press, Redmond, WA, USA.

Pike, R., Dorward, S., Griesemer, R., and Quinlan, S. (2006). Interpreting the data: Parallel analysis with Sawzall (draft). *Scientific Programming*, **13**(4), 277–298.

Plishker, W., Ravindran, K., Shah, N., and Keutzer, K. (2004). Automated task allocation for network processors. In *Proceedings of Network System Design Conference*, pages 235–245.

Protegé (2010). Protegé Ontology Library. http://protegewiki.stanford.edu/index.php/. Date accessed: Nov. 2009.

Qadeer, S. and Wu, D. (2004). KISS: Keep it simple and sequential. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'04)*, volume 39-6, pages 14–24, New York, NY, USA. ACM Press.

Rajan, H., Kautz, S. M., and Rowcliffe, W. (2010). Concurrency by modularity: Design patterns, a case in point. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10)*, pages 790–805, New York, NY, USA. ACM.

Randall, K. H. (1998). *Cilk: Efficient Multithreaded Computing*. Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Red Gate Software (2008). Ants. www.red-gate.com. Date accessed: Mar. 2008.

Robilliard, D., Marion-Poty, V., and Fonlupt, C. (2008). Population parallel GP on the G80 GPU. In *Proceedings of the 11th European conference on Genetic programming (EuroGP'08)*, pages 98–109, Berlin, Heidelberg. Springer-Verlag.

Robison, A. D. and Johnson, R. E. (2010). Three layer cake. In *Proceedings of the Workshop on Parallel Programming Patterns (ParaPLOP'10)*.

Rodriguez, J., Cesario, C., Galvan, K., Gonzalez, B., Kroner, G., Rutigliano, G., and Wilson, R. (2005). *IBM rational application developer v6 portlet application development and portal tools*. IBM Corp., Riverton, NJ, USA.

Roscoe, A. W., Hoare, C. A. R., and Bird, R. (1997). *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Rossi, F. V. (2010). *Graphics hardware accelerated transmission line matrix procedures*. Master's thesis, Department of Electrical and Computer Engineering, University of Victoria.

Rothlisberger, D., Harry, M., Villazon, A., Ansaloni, D., Binder, W., Nierstrasz, O., and Moret, P. (2009). Senseo: Enriching Eclipse's static source views with dynamic metrics. In *Procceddings of the IEEE International Conference on Software Maintenance*, volume 0, pages 383–384, Los Alamitos, CA, USA. IEEE Computer Society.

Royce, W. W. (1970). Managing the development of large software systems: concepts and techniques. In *Proceedings of the IEEE WESTCON*, Los Angeles, CA, USA. IEEE Computer Society Press. Reprinted in Proceedings of the International Conference on Software Engineering (ICSE) 1989, ACM Press, pp. 328-338.

Russell, R. M. (1978). The CRAY-1 computer system. *Communications of the ACM*, **21**(1), 63–72.

Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W. W. (2008). Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, pages 73–82, New York, NY, USA. ACM.

Schmidt, D., Stal, M., Rohnert, H., and Buschmann, F. (2001). *Pattern-Oriented software architecture—Patterns for concurrent and networked objects*. Wiliey.

Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., and Hanrahan, P. (2008). Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, **27**(3), 18:1–18:15.

Shah, N., Plishker, W., and Keutzer, K. (2003). NP-Click: A rrogramming model for the Intel IXP1200. In *Proceedings of the Workshop on Network Processors (NP-2) at the International Symposium on High Performance Computer Architecture (HPCA-9)*.

Shvaiko, P. and Euzenat, J. (2008). Ten challenges for ontology matching. In *Proceedings of the OnTheMove Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part II on On the Move to Meaningful Internet Systems (OTM '08)*, pages 1164–1182, Berlin, Heidelberg. Springer-Verlag.

Simmons, M. L. and Wasserman, H. J. (1988). Performance comparison of the Cray-2 and Cray X-MP/416 supercomputers. In *Proceedings of the 1988 ACM/IEEE conference on Supercomputing (SC'88)*, pages 288–295, Los Alamitos, CA, USA. IEEE Computer Society Press.

Singh, N. (2011). *Integrated Tooling Framework for Software Configuration Analysis*. Master's thesis, Department of Computer Science, University of Victoria.

Singh, N., Gibbs, C., and Coady, Y. (2007). C-CLR: A tool for navigating highly configurable system software. In *Proceedings of the Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS'07)*, New York, NY, USA. ACM.

Snir, M. and Otto, S. (1998). *MPI–The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA.

Stamatakis, A. and Ott, M. (2008). Exploiting fine-grained parallelism in the phylogenetic likelihood function with MPI, PThreads, and OpenMP: A performance study. In *Proceedings of the IAPR International Conference on Pattern Recognition in Bioinformatics (PRIB'08)*, pages 424–435, Berlin, Heidelberg. Springer-Verlag.

Star Quotes (2010). Seymour Cray quotes. http://www.quotesstar.com/people/occupations/seymour-cray-quotes.html. Date accessed: Dec. 2010.

Stroustrup, B. (2000). *The C++ Programming Language*. Addison-Wesley Longman Publishing, Boston, MA, USA, 3rd edition.

Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *A Dr. Dobb's Journal*, **30**(3).

Tanenbaum, A. S. (2007). *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA.

TinyOS (2009). TinyOS Home Page. http://webs.cs.berkeley.edu/tos/. Date accessed: Mar. 2009.

Total View Technologies (2008). Total View Technologies. www.totalviewtech.com. Date accessed: Mar. 2008.

Tu, Q. and Godfrey, M. W. (2002). An integrated approach for studying architectural evolution. In *Proceedings of the International Workshop on Program Comprehension (IWPC'02)*, pages 127–136. IEEE Computer Society Press.

Tun, N. N. (2006). Semantic enrichment in ontologies for matching. In *Proceedings of the Australasian workshop on Advances in ontologies (AOW '06)*, volume 72, pages 91–100, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

von Behren, R., Condit, J., and Brewer, E. (2003). Why events are a bad idea (for high-concurrency servers). In *Proceedings of the Conference on Hot Topics in Operating Systems (HOTOS'03)*, pages 4–4, Berkeley, CA, USA. USENIX Association.

von Neumann, J. (1988). The principles of large-scale computing machines. *IEEE Annals of the History of Computing*, **10**(4), 243–256.

WC3 – Semantic Web (2004). RDF (Resource Description Framework). Date accessed: Nov. 2009.

Weiser, M. (1984). Program slicing. *IEEE Transactions on Software Engineering*, **10**(4), 352–357.

White, T. (2007). Running Hadoop MapReduce on Amazon EC2 and Amazon S3. *Amazon Web Services Developer Connection*.

Williams, A., Thies, W., and Ernst, M. D. (2005). Static deadlock detection for Java libraries. In *Proceedings of the European Conference on Object-Oriented Programmign (ECOOP'05)*, pages 602–629.

WinMerge (2011). WinMerge. http://winmerge.org/. Date accessed: Feb. 2011.

Xu, M., Bodik, R., and Hill, M. D. (2003). A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the International Symposium on Computer Architecture (ISCA'03)*, pages 122–135, New York, NY, USA. ACM.

Xu, M., Malyugin, V., Sheldon, J., Venkitachalam, G., Weissman, B., and Inc, V. (2007). Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the Annual Workshop on Modeling, Benchmarking and Simulation (MoBS'07)*.

Zöbel, D. (1983). The deadlock problem: A classifying bibliography. *SIGOPS Operating Systems Review*, **17**, 6–15.