2010

FraSPA: A Framework for Synthesizing Parallel Applications

Ritu Arora
University of Alabama at Birmingham

FRASPA: A FRAMEWORK FOR SYNTHESIZING PARALLEL APPLICATIONS

by

RITU ARORA

PURUSHOTHAM BANGALORE, COMMITTEE CHAIR
IOANA BANICESCU
JEFF GRAY
MARJAN MERNIK
ANTHONY SKJELLUM

A DISSERTATION

Submitted to the graduate faculty of The University of Alabama at Birmingham,
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

BIRMINGHAM, ALABAMA

2010

FRASPA: A FRAMEWORK FOR SYNTHESIZING PARALLEL APPLICATIONS

RITU ARORA

COMPUTER AND INFORMATION SCIENCES

ABSTRACT

Scientists, engineers and other domain-experts have computational problems that are growing in size and complexity, thereby, increasing the demand for High Performance Computing (HPC). The demand for reduced time-to-solution is also increasing and simulations on high performance computers are being preferred over physical prototype development. Though HPC is gradually becoming indispensible for business growth, the programming challenges associated with HPC application development are a key bottleneck to embracing it on a massive scale. Current high-level approaches for generating HPC applications are either domain-dependent or do not leverage from existing applications.

Message Passing Interface (MPI) is the most popular standard for writing parallel applications for distributed memory HPC platforms. The development of parallel applications using MPI often begins with working sequential applications that undergo major rewrites to incorporate appropriate calls to MPI routines. Writing efficient parallel applications using MPI is a complex task due to the extra burden on programmers (including domain-experts) to manually and explicitly handle all the complexities of message-passing (*viz.,* data distribution and load-balancing). Invasive manual reengineering of existing applications is also required for making them checkpointed to overcome resource-failures in distributed environments.

A Framework for Synthesizing Parallel Applications (FraSPA) has been developed in this research with the goal of reducing the complexities associated with the process of developing checkpointed message-passing applications. FraSPA is capable of doing automatic code instrumentation for parallelization and checkpointing on the basis of the high-level specifications provided by the end-users. The high-level specifications are provided by domain-specific languages developed in this research. For the selected test cases, there is more than 90% of reduction in the end-user effort in terms of the number of lines of code written manually while requiring no explicit changes to the existing code. The performance of the generated code is within 5% of that of the manually-written code. FraSPA was developed using a combination of modern software engineering techniques (*viz.* generative programming and model-driven engineering) and has the potential of being extended to support heterogeneous architectures, multiple programming languages, and various parallel programming paradigms.

DEDICATION

Gururev Brahma, Gururev Vishnu,

Gururev Devo, Maheshwara,

Gururev Shakshat Param Brahma,

Tasmayi Shree Guruveh, Namoh Namah!

This work is dedicated to my advisor, Dr. Purushotham Bangalore – Thanks for imparting the knowledge that I needed to be where I want to be in life!

ACKNOWLEDGEMENTS

I am very grateful to my advisor, Dr. Purushotham Bangalore, for accepting me as his Ph.D. student and giving me a chance to reach this far. Because of his liberal and progressive approach, he has created a positive impact on my personality such that today, I am confident of being a successful researcher in both collaborative and independent research environments. With his immense knowledge in the field of High Performance Computing, he has very calmly and patiently guided me in the Ph.D. program. I shall always be indebted to him for his able guidance that made this work possible.

I am very thankful to my role-model, Dr. Jeff Gray, for providing me the required training on modern software engineering techniques through his courses. His timely support and able guidance on several issues has helped me in accomplishing my research and career goals. I am especially thankful to him for the opportunity to collaborate with the members of his laboratory and mentoring me on the art of writing high-quality research papers. It was a great learning experience to work with him and I shall always strive to match-up to his standards.

I am very grateful to Dr. Marjan Mernik for mentoring me in the area of Domain-Specific Languages. He has taken great interest in my work and has always been available to advise me. I am extremely grateful to him for all the efforts that he has made in helping me understand the topics that I have found difficult and in improving the quality of our papers. I am very thankful to him for his valuable feedback in improving this dissertation.

I am thankful to Dr. Ioana Banicescu and Dr. Anthony Skjellum for very kindly accepting the invitation to serve on my graduate study committee and for providing their valuable feedback for improving this dissertation.

I would like to thank Dr. Frédéric Jouault for letting me attend his lectures on model-driven engineering and helping me understand the basics of the AMMA platform used in this research. I am grateful to my colleague and mentor, Dr. Suman Roychoudhury, for guiding me through the initial work done for this research – thanks for understanding me and being there for me during tough times! My expertise-level in RSL and PARLANSE wouldn't have been the same without his help. I am sincerely thankful to Dr. Ira Baxter for his timely help on the questions related to DMS. He has very patiently understood my questions and has answered them in great depth and with care. I am grateful to my mentor, Dr. Kai Shen, for showing me the new directions in which this dissertation research can be applied.

I would like to thank Dr. Elliot Lefkowitz and all the members of his laboratory – especially Jim Moon, Curtis Hendrickson, Don Dempsey, and Catherine Galloway – for giving me the opportunity to work them on the Viral Bioinformatics Resource Project. The work experience that I gained through my association with them has made me a well-rounded software engineer and has given me the required exposure to the American work-culture. I have not only improved my attention-to-detail by working with them but have also learnt the art of demystifying complex problems.

I am thankful to all the faculty members of the Computer and Information Sciences Department at UAB for the courses they have offered. Their courses have helped me in broadening my knowledge and in passing the qualifying exams, without which it would have not been possible to advance in the graduate program.

I am extremely grateful to my colleague and friend, Dr. Vetria Byrd, for her incessant encouragement and company in tough times – you have been such a great listener and my rock-solid-support-system! To my other colleagues at UAB - Yu Sun, Zekai Demirezen, Ferosh Jacob, and Saraswathi Mukkai – I enjoyed working with you all. I will carry the wonderful memories of the time we spent together working on tough and not-so-tough problems for the rest of my life. I am very grateful to Janet, Kathy, and John for their kind help, care and support. Janet and Kathy have always been there for me as great listeners and guides. This dissertation work would not have been complete without the timely support from the IT staff in the Computer and Information Sciences Department and I am very thankful to them. I would also like to thank UAB and the Computer and Information Sciences Department for the computing resources and for employing me as a graduate assistant for majority of the duration of my graduate study.

I am grateful to Dr. David Young at the Alabama Supercomputing Center for helping me in using their computational resources. I am grateful to my counselor, Consuelo Click, for her kind help and support. I appreciate the kind assurance and advice from Dr. Jeff Engler and Susan Banks – it meant a lot to me! In the end, I am thankful to

my family for keeping me motivated and helping me overcome all the hurdles during the course of my graduate study – thank you for your financial, moral, and spiritual support!

TABLE OF CONTENTS

Page

TABLE OF CONTENTS (Continued)

Page

CHAPTER

TABLE OF CONTENTS (Continued)

Page

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ALC	Application-Level Checkpointing
ANT	Another Neat Tool
AMMA	ATLAS Model Management Architecture
AOP	Aspect-Oriented Programming
API	Application Programming Interface
AST	Abstract Syntax Tree
ATL	Atlas Transformation Language
CaR	Checkpointing and Restart
CBIR	Content Based Image Retrieval
CFD	Computational Fluid Dynamics
CPU	Central Processing Unit
DALC	DSL for Application-Level Checkpointing
DMS	Design Maintenance System
DPnDP	Design Patterns and Distributed Process
DSL	Domain-Specific Language
DSM	Domain-Specific Modeling
EBNF	Extended Backus-Naur Form
EMF	Eclipse Modeling Framework
EMOF	Essential Meta-Object Facility

FraSPA	Framework for Synthesizing Parallel Applications
GA	Genetic Algorithm
GPL	General-Purpose Language
GPGPU	General-Purpose Graphical Processing Unit
GUI	Graphical User Interface
Hi-PaL	High-Level Parallelization Language
Hi-Spade	Hierarchy-Savvy parallel algorithm design
HPC	High Performance Computing
ISC	Invasive Software Composition
LOC	Lines of Code
MAP_3S	MPI Advanced Pattern-Based Parallel Programming System
MDE	Model-Driven Engineering
MIMD	Multiple Instruction Multiple Data
MOF	Meta-Object Facility
MPI	Message Passing Interface
OCL	Object Constraint Language
PARLANSE	PARallel LANguage for Symbolic Expressions
PTE	Program Transformation Engine
RMS	Root Mean Square
RSL	Rule Specification Language
SQL	Structured Query Language
SSC	Source-to-Source Compiler
TCS	Textual Concrete Syntax

CHAPTER 1

INTRODUCTION

With the advancement in science and technology, the computational problems are growing in size and complexity, thereby, resulting in the increase in the demand for High Performance Computing (HPC) resources. To keep up with the competitive pressure, the demand for reduced time-to-solution is also increasing and simulations on high performance computers are being preferred over physical prototype development and testing. Recent studies have shown that though HPC is gradually becoming indispensible for business growth, the programming challenges associated with the development of HPC applications (*e.g.,* lack of HPC experts, learning curve and system manageability) are key deterrents that stop companies from embracing HPC on a massive scale. Therefore a majority of companies are stalled at the desktop-computing level [1, 2]. The challenges associated with the development of HPC applications are further elaborated in Section 1.1.

1.1 Challenges in HPC Application Development

The general challenges associated with the HPC application development are presented in this section. These challenges were the main motivating factors behind the research done for this dissertation and were found across several application-domains.

1.1.1 Problem of Plenty

Scientific (or HPC) applications are often written in C/C++ or FORTRAN and are run on HPC platforms using a parallel programming paradigm. Because there are multiple types of HPC platforms available today, there are multiple parallel programming paradigms available, each best-suited for a particular platform (or architecture). For example, Message Passing Interface (MPI) [3] is best suited for developing parallel programs for distributed memory architectures, whereas, OpenMP [4] is widely used for developing applications for shared memory architectures. It is a difficult task to write portable and performance-oriented parallel programs that are scalable across multiple HPC platforms and the programmers are often required to reengineer their applications as per the underlying HPC architecture. This challenge is closely related to the challenges presented in Sections 1.1.2 and 1.1.3.

1.1.2 Modern Computing Platforms

The tremendous progress in the computer architecture discipline over the last few decades has lead to the development of fast personal computers that are capable of providing theoretical peak performance of more than 100 gigaFLOPS - equaling the performance of some of the advanced supercomputers from about a decade ago [5, 6, 7]. Even though the advancement in the area of computer architecture has resulted in such high theoretical peak performance for the latest personal computers, it is difficult to effectively exploit the full potential of these architectures due to the slow rate of advancement in the area of parallel programming environments. To understand this problem at the grass root-level, one must realize that modern computers are immensely

complex and have different characteristics as compared to their predecessors such that the process of determining the performance of applications on modern architectures has also undergone a transition [7]. As noted in [5], on today's modern architectures, loading and storing the data in memory can be slower (200 clocks) than doing a multiply operation (4 clock cycles). However, in the previous years, the speed of doing a multiply operation was considered as an important performance characteristic. Modern day platforms' microarchitectural features (*viz.* memory hierarchy, register sets, and special instruction sets) crucially determine the performance of an application, which implies that, with every new HPC platform, the programmer must re-optimize the application to achieve maximum performance. A code that is optimal for a particular architecture might not depict the same performance on a different architecture. If the highest performance is required, the applications are likely to be hand-tuned and hand-written in assembly language (example, Intel's Integrated Performance Primitives) [7]. It is expensive to re-optimize, re-tune or re-implement hand-written code to adapt it for the latest architecture.

A gradual shift from homogeneous architectures to complex heterogeneous architectures is also being observed [5]. The combination of CPUs, cell processors, field-programmable gate arrays, and graphical processing units is being touted as the next evolution in HPC (*e.g.,* IBM's Roadrunner and ORNL's Jaguar). At the core of this evolution is Moore's Law, which has accurately predicted for over three decades that the density of transistors on a chip will double every 18 months. Thus, modern multi-core and many-core architectures feature hundreds of cores on a chip [5]. The heterogeneity in modern architectures and the constant increase in the number of processors in a parallel system have lead to complex systems with a short Mean Time Between Failures (MTBF)

[7]. The execution time of computational science applications running on such complex systems might be greater than the MTBF of the underlying resources. It is therefore imperative to develop a portable and distributed fault-tolerance mechanism to support the parallel applications developed for such complex systems [8]. The fault-tolerance mechanism would save time in restarting the application in the event of any failure in the underlying platform. The applications are migrated from the failed resource to a healthy one, and are restarted from the latest saved state instead of being restarted from scratch. Due to memory constraints, it is important to avoid taking the core dumps of the execution states of the applications. A lean-and-mean approach for fault-tolerance is therefore essential.

1.1.3 Predicament of Programmers

It is also important to understand the role of key players in the process of HPC application development. HPC applications are traditionally developed by domain-experts and computer scientists. The domain-experts, as being referred to here, are the scientists who lack formal training in computer science discipline but are likely to have engineering, physics, chemistry, or biology backgrounds. They tend to be researchers who are more interested in achieving accurate results than in learning how the applications were developed [9]. Due to lack of access to computer scientists (or HPC experts), researchers often develop their own applications and spend quality time to learn new programming paradigms or to understand the latest architectures [1, 2]. They seldom have time to hone their performance-programming skills. On the other hand, computer scientists are from the traditional computer science background and are knowledgeable

about programming languages and computer architectures. Yet, to develop the optimal solution for the scientific problem at hand, they must develop an understanding of the problem domain and must accurately interpret the requirements of domain-experts. When working in conjunction with a computer scientist, a domain-expert is responsible for interpreting the results and mapping the source of any errors to either the specifications or the code developed by the computer scientist. Therefore, domain-experts and computer scientists must climb the learning curve and spend quality time before developing scalable and performance-oriented applications that give accurate results. It has also been observed that many programmers implement and optimize similar functionality for multiple platforms without considering the reusability-quotient of their applications [7, 9]. As explained earlier, they often repeat the process of application implementation, optimization and tuning when a new architecture emerges [7]. In short, HPC application development has emerged as an interdisciplinary task requiring that the programmer be knowledgeable not only in algorithms and programming languages but also in computer architectures [7].

1.1.4 Summary of Challenges and Discussion

A summary of the challenges presented so far in this chapter is as follows:

1. There are multiple parallel programming platforms and hence multiple parallel programming paradigms. Each programming paradigm has a learning curve associated with it.

2. It is increasingly hard to harness the peak performance provided by the modern HPC platforms due to the slow rate of advancement in the area of parallel programming environments.
3. Adapting the applications to new architectures is a time-consuming activity because it might require re-tuning, re-optimization, or re-implementation.
4. HPC application development has become an interdisciplinary task requiring that programmers not only have the knowledge of programming languages, algorithms, and architectures but also clearly understand the problem domain [7].
5. The heterogeneity in architecture and the constant increase in the number of processors in HPC platforms are likely to produce complex parallel computing platforms with short MTBF. The execution time of the applications running on such platforms might be longer than the MTBF of the platform and therefore, a fault-tolerance mechanism is required.

In the light of the aforementioned challenges related to fast changing, increasingly complex, and diverse computing platforms, key questions that arise are:

1. Is it feasible to achieve portability and optimal performance with reasonable effort?
2. Can efficient parallel programs be automatically generated by computers?
3. Can we bring scalability and performance to domain-experts in the form of parallel computing without any need to learn low-level parallel programming?
4. Can we facilitate the transition of HPC from the realms of specialized and scientific application development into mainstream business?

5. Can we mitigate the negative impact of the reduced MTBF of the complex parallel computing platforms on the execution time of the applications?

Most of these questions are associated with the accidental complexities related to the HPC application development process while some questions can be mapped to the essential complexities. Fred Brooks [10] identified two complexities associated with the application development process – essential and accidental. Essential complexities are related to the problem space and are deep-seated domain challenges [10]. Accidental complexities are related to the solution space, *i.e.,* the tools and techniques used for implementing the solution to a problem. Brooks also mentioned in [10] that there is no single silver bullet (or no single approach) to alleviate all the complexities associated with the software development process. Therefore, only through an effective combination of multiple modern software engineering techniques, can one attack the challenges associated with the development of HPC applications.

1.2 Scope of Research and Research Statement

While geared towards tackling the challenges associated with the development of HPC applications, this dissertation's main focus was to address the aforementioned challenges in the context of distributed memory architectures. The rest of this chapter will therefore provide an overview of the niche area of this dissertation, which is, cost-effective explicit parallelization for distributed memory architectures.

HPC applications for distributed memory architectures can either be developed using methods of implicit parallelization or explicit parallelization. Implicit parallelization is achieved by using pure implicitly parallel languages (*e.g.,* X10 [11],

Orca [12], SISAL [13], Fortress [14]). The parallelism is characteristic of the language itself and therefore the language compiler or interpreter is able to automatically parallelize the computations on the basis of the language constructs used. This method of parallelization enables the programmer to focus on the problem to be solved instead of worrying about the low-level details of how the parallelization is achieved. However, there are some disadvantages associated with this mode of parallelization and some of them are: the code must be developed from scratch, existing legacy applications written in C/C++/FORTRAN cannot benefit from this approach, programmer might have limited flexibility to experiment with different algorithm-design options, and debugging is difficult because it is unclear which code construct might be causing performance loss [15].

Explicit parallelization is achieved by using specialized libraries for parallelization in conjunction with the programming language of the programmer's choice. With this approach the programmer inserts the library calls in the existing application at the points where parallelization is desired. This approach gives substantial amount of flexibility to the programmer by letting them make a choice about the portions of their program that should run in parallel and the way they should be parallelized. This approach also helps the programmer to leverage from their existing sequential applications. Though this approach is very popular due to the control, flexibility, and performance it provides, it puts the burden of parallelization on the programmer in terms of time and effort required to achieve the goals. The programmer is responsible for understanding the parallelism in his application and expressing it intelligently in order to gain maximum performance. Due to its advantages in terms of performance, the focus of

this dissertation is on explicit parallelization. The specific advantages and disadvantages of explicit parallelization in context of this dissertation are further explored in the following paragraphs.

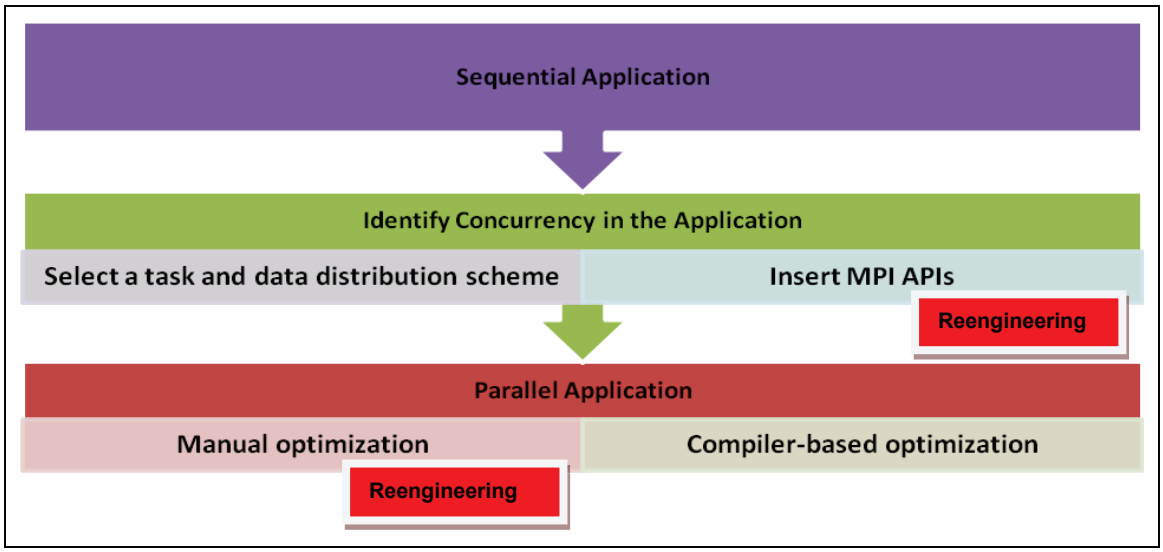

Figure 1-1 - Usual process of writing a parallel program using MPI

Under the category of explicit parallelization, MPI is regarded as the most popular standard for writing portable and scalable parallel applications for distributed memory architectures by embedding calls to MPI-routines in sequential applications. The process of developing a parallel application using MPI is pictorially depicted in Figure 1-1. As can be noticed from Figure 1-1, often, programmers develop parallel applications using MPI by taking working sequential applications, identifying concurrency in them, and then expressing the concurrency in terms of data or task distribution amongst the available processors. In order to express the concurrency explicitly, the programmers often restructure the existing sequential applications and insert calls to MPI-routines. The programmers have to bear the burden of explicitly mapping the tasks to the processors, manually orchestrating the exchange of messages, load-balancing and synchronization. The parallel version generated by inserting the MPI-routines in the sequential application

is further optimized as per the machine architecture, to obtain maximum efficiency or speedup. The most common optimization techniques are related to arrays and memory management, loops, arithmetic operations, and data input or output. The manual optimization of the code might involve several iterations of code changes. Overall, developing, debugging, and maintaining parallel programs using MPI is a challenging task. The process of explicit parallelization using the MPI standard eventually becomes an intrusive reengineering activity that puts extra burden on programmers to handle too many low-level details manually (including handling errors and race conditions). Also, the MPI layer provides a poor level of abstraction as it deals with explicit buffers and message transfers and therefore exposes data structure details to the programmer [16, 17].

Not only is manual intrusive reengineering for explicit parallelization a complex and error-prone activity involving critical resources (*viz.* time and effort), but it also makes code maintenance difficult due to the cross-cutting concerns [18]. Cross-cutting concerns are the concerns that are spread across multiple methods within multiple modules of an application [18]. These cross-cutting concerns lead to scattered or tangled code. In order to make a single change in a cross-cutting concern, it becomes necessary to replicate the changes at multiple places. Examples of cross-cutting concerns in parallel applications are communication, synchronization, load-balancing, and checkpointing.

The software development process using explicit parallelization leaves little scope for code reuse because it involves ad-hoc design decisions [17]. In case a programmer wants to set-up communication between the processors, there are multiple options available, each with specific trade-offs (*e.g.,* synchronous/asynchronous, point-to-point/one-sided/collective) but there are no well-established rules or design patterns to

select one option over the other. There are some mechanical steps for setting up the MPI environment that can be found in every MPI program. Analyses of code samples from diverse domains (see Figure 1-2) also show replicated code constructs for tasks other than setting-up the MPI environment. Such commonalities (replicated code constructs) indicate that there is definitely a scope of reducing the effort involved in developing HPC applications by promoting code reuse.

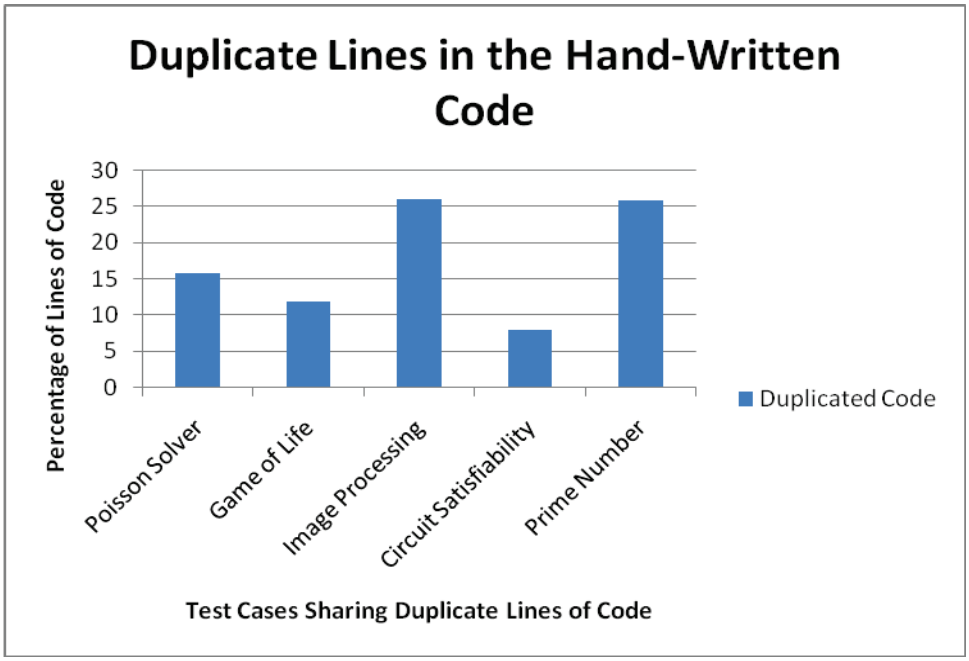

Figure 1-2 – Percentage of duplicate lines of code that applications share

Despite the challenges mentioned in this chapter, MPI is the most widely used standard for writing parallel applications and it has been implemented for several distributed memory architectures. The main advantages of MPI are its speed and portability. Hence, to effectively exploit the HPC power of low-cost distributed memory architectures, parallel programming based on the most widely used parallel programming standard (*i.e.,* MPI) should be made less complex (through abstractions). In the light of the aforementioned issues, the main goal of this research was to raise the level of

abstraction of parallel programming using MPI, such that, the effort involved in developing a parallel application by manually reengineering an existing sequential application, is significantly reduced. The reduction in effort can be quantified in terms of the reduction in the number of lines of code the programmer has to write manually. The generative programming [19] approach adopted in this research was helpful in developing a framework that employs reusable code components for synthesizing parallel programs for a wide range of application-domains. This framework, called FraSPA (**Fra**mework for **S**ynthesizing **P**arallel **A**pplications), is useful for developing parallel applications for distributed memory models without the burden of learning or using MPI. However, the programmers using FraSPA still need to identify concurrency in the application and express it in a very succinct manner. Therefore, FraSPA employs a user-guided approach to synthesize optimized parallel programs from existing sequential programs.

A high-level, declarative and platform-independent Domain-Specific Language (DSL) called High-level Parallelization Language (Hi-PAL) has been developed in this research for obtaining the concurrency-specifications from the programmers. A set of guidelines can be provided to the end-users for helping them in expressing the concurrency in their applications through Hi-PaL. These guidelines would also be useful for explaining the key steps for analyzing the sequential application and the best practices to achieving a parallel code with high performance. In summary, in order to automatically synthesize parallel applications through FraSPA, the programmers are required to:

- understand the concept of concurrency,

- follow the guidelines provided for expressing concurrency, and
- install the required tools on their machines.

Thus, with a minimum investment, the programmers can synthesize parallel applications using FraSPA without getting involved in the complexities associated with MPI. This research, therefore, raises the level of abstraction of the scientific application development process. If the programmers wish to make the synthesized applications checkpointed (one of the techniques required for making the applications fault-tolerant), they can do so with the help of the DSL for Application-Level Checkpointing (DALC) that was developed as a part of this research. Similar to the process of generating parallel applications on the basis of the Hi-PaL specifications provided by the end-user, FraSPA uses the DALC specifications to make the existing applications checkpointed. The FraSPA generates the desired code for parallelization and checkpointing for a wide range of applications using reusable code components, design-templates, program transformation system, domain-specific languages, and glue code. The details of the implementation of the framework are provided in Chapter 3. FraSPA can be extended to provide support for additional functionality and helps in the incremental development of applications with multiple alternatives. In short, FraSPA bolsters the claim made by the following research statement:

"An extensible and flexible framework can be developed for non-invasively synthesizing scalable, MPI-based, performance-oriented and checkpointed parallel applications in a user-guided manner with the goal of reducing the complexities associated with explicit parallelization without compromising the performance or accuracy of results."

1.3 Key Contributions

The key contributions made to the field of HPC through the design and implementation of FraSPA are summarized in this section. These contributions are the solutions to some of the challenges related to the HPC application development in general, and explicit parallelization using MPI in particular. FraSPA,

1) Brings performance and scalability to domain-experts in the form of parallel computing without the need to learn low-level parallel programming or to do intrusive reengineering.
2) Separates parallel and sequential concerns to reduce the code complexity and improve the maintainability of the application.
3) Promotes code reuse and code correctness through the usage of design-templates.
4) Provides support for generating checkpointed applications that can be used in combination with resources for fault-detection in order to develop fault-tolerant solutions.
5) Can parallelize applications from diverse domains (*e.g.,* image processing, computational fluid dynamics, and evolutionary algorithms).
6) Increases the programmer productivity in terms of the decrease in the number of lines of code written manually.
7) Reduces the time-to-solution due to the reusable nature of its code components.

In summary, FraSPA hides the challenges associated with the low-level parallel programming from the domain-experts. It also helps them by reducing the time involved in parallelizing their applications by utilizing reusable code components.

1.4 Broader Impact

FraSPA demonstrates a methodology for composing optimized HPC applications from reusable components and hence is an ideal example of amalgamation of modern software engineering techniques with HPC application development. This research has the potential of bridging the complexity gaps between scientific application development and complex hardware platforms [5]. FraSPA can be extended to provide support for multiple parallel programming models (*e.g.,* support for synthesizing parallel applications for shared memory paradigms and multi-core architectures) and hence multiple parallel programming platforms. Apart from extending FraSPA to provide support for multiple programming paradigms, it can also be extended to support the automatic parallelization of sequential applications written in other legacy languages (*e.g.,* FORTRAN) and dialects. The likelihood of major manual rewrites in the event of any change in the application requirements, HPC platform, or implementation algorithms, is speculated to decrease with the usage of the approach presented in this dissertation. FraSPA has the potential of lowering the barriers to the adoption of HPC [1, 2] by domain-experts who do not have any exposure to low-level parallel programming. FraSPA can also be adopted by instructors for teaching the process of developing MPI-based parallel programs incrementally and at a conceptual-level before going into the low-level details of MPI-programming. The overall principle behind the working of FraSPA can be applied to develop a domain-specific modeling language for specifying parallel computations in specific domains such that the domain-experts can provide the specifications once and generate code in several base languages including those for implicit parallelization (*viz.*

X10, Fortress, or SISAL) by using language-specific interpreters.

The checkpointing mechanism developed in this research is useful for making both parallel and sequential applications checkpointed and can be incorporated with the strategies for fault-detection to develop fault-tolerant solutions. This approach has the potential of being adapted to develop fault-tolerant applications for multi-core and many-core architectures as well. In future, if the support for fault-tolerance is available as a part of the MPI library, the approach developed in this research can be used for automatically and non-invasively reengineering the existing parallel applications for embedding the calls to the latest MPI routines. As a consequence of this research, stronger interactions of HPC researchers, software engineers, and scientists can be fostered across different domains.

1.5 Overview of the Dissertation

The background and the related research work are discussed in Chapter 2 of this dissertation. The Generative Programming [19] tools and techniques that were used for implementing FraSPA are also described in detail in Chapter 2. These techniques obviate some of the barriers that scientists face in adopting high-level abstractions. Chapter 3 is related to the design and development of FraSPA. The case-studies and results are discussed in Chapter 4. The potential future work is presented in Chapter 5 and conclusion is presented in Chapter 6.

CHAPTER 2

BACKGROUND AND RELATED WORK

This dissertation demonstrates the effective application of modern software engineering techniques to automate the process of HPC application generation. The main tangible contribution of this research is a framework that enables the automatic synthesis of MPI-based, checkpointed parallel applications from high-level specifications and existing sequential applications. The framework is implemented by using the combination of Generative Programming (GP) techniques [19] and a set of Domain-Specific Languages (DSLs) [20]. Design-templates have also been used in this research to capture the most commonly used data distribution, communication, and synchronization patterns in MPI-based programs. Model-Driven Engineering (MDE) [21] was used to make the framework extensible and flexible. Application-Level Checkpointing (ALC) technique was used to make the generated applications fault-tolerant [22].

This chapter will provide a background discussion of the technologies used in this dissertation research and the related work. Section 2.1 presents a discussion on Aspect-Oriented Programming (AOP) [18] and its usage in the initial work done for this research. Section 2.2 gives a brief overview of an alternative to AOP-based approach for program synthesis, which is called, Invasive Software Composition (ISC) [23]. The generative programming tools and techniques used in this research are explained in

Section 2.3. DSLs are explained in Section 2.4. MDE and its usage in this research are explained in Section 2.5. The checkpointing mechanism for making the HPC applications fault-tolerant is explained in Section 2.6. A discussion of the work related to high-level parallel programming is presented in Section 2.7. A discussion of the work related to checkpointing (and hence fault-tolerance) is presented in Section 2.8. A general discussion is provided in Section 2.9.

2.1 Aspect-Oriented Programming

When a concern or functionality is spread across multiple methods within multiple modules of an application (*e.g.,* profiling and logging), it is known as a crosscutting concern [18]. Such concerns are difficult to modularize using traditional languages because each concern is scattered across modularity boundaries, making it hard to maintain and reuse. To make a single change in a crosscutting concern, it is necessary to replicate the changes at multiple places. Therefore, crosscutting concerns lead to tangled or scattered code [18] and it is advantageous to capture the functionality of each concern in a separate module.

AOP offers a new modular construct that cleanly separates crosscutting concerns (or secondary functionality) from the core computations, thereby leading to an improved quality of code in terms of improved cohesion, coupling, and code reuse. A crosscutting concern is isolated in a modular unit, called aspect, which can be woven into an application as needed. Each aspect, thus, leads to the localizing of the description of a crosscutting concern in a single place. It should be noted that, by itself, AOP is only a concept. There are several AOP languages that materialize the AOP concepts and extend

the traditional programming languages. AOP languages (*e.g.,* AspectC++ [24] and AspectC [25]) overcome the limitations of traditional programming languages by extending them with constructs for programming aspects. The programmers write a base code in a traditional language (*e.g.,* C++ or C) and aspect code in the relevant AOP language (*e.g.,* AspectC++ or AspectC) so that the aspect code affects the execution of specific code in the base program. The aspect code is combined with the base program with a tool called *aspect weaver* that finally generates the transformed code. The transformed code has the desired functionality (as specified in the aspect code) at the desired number of places added to the base program. It can be compiled and run like a normal program. This weaving process is pictorially depicted in Figure 2-1.

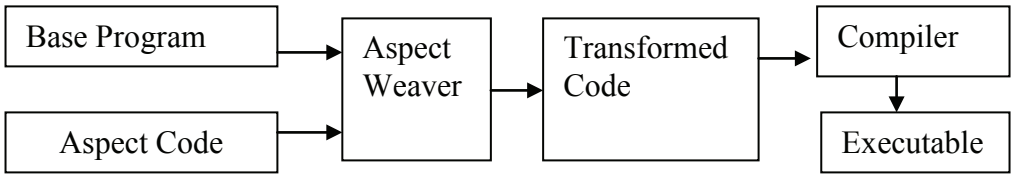

Figure 2-1: Aspect weaving process

Some of the common constructs in AOP languages are as follows:

- **Join Point**: A location in a program where a crosscutting concern emerges - for example, method call and method execution.
- **Match Expression**: A search pattern in string format. It may specify the type, namespace, class, function, or template in the base program which should be used as a handle for weaving the aspect code.
- **Pointcut**: Determines the condition on which the aspect code would be executed. It is a "set of join points and are described by a pointcut expression" [18].

- **Pointcut Expression**: Composed of match expressions (that are used to find the set of join points), pointcut functions (for filtering specific join points), and algebraic operators (used for combining pointcuts).
- **Advice**: Captures the implementation of a crosscutting concern and defines actions to be performed at associated join points. It can also be used to introduce a new function or an attribute or a type at join point and is of the types: `before`, `after`, and `around`.
- **Aspect**: Implements a crosscutting concern and hence, localizes its functionality in a separate module. An aspect in AspectC++ is similar to a class in C++ and is defined by pointcut expressions and advice. It may contain attributes, methods, and advice declarations. Like classes, an aspect can inherit from other classes and aspects.

Consider the C++ code shown in Figure 2-2 that does simple addition, subtraction, multiplication and division. It is desired to print the signature of every method in Figure 2-2 for debugging and testing purposes without making any changes to the existing code (shown in Figure 2-2). This can be achieved through a printing aspect (`aspect printing` at line # 4 of Figure 2-3) written using AspectC++ and shown in Figure 2-3. As can be noticed from line # 5 of the code in Figure 2-3, this printing aspect prints the method signature before the method is executed. In general, the '`*`' represents a wildcard, and '`...`' represents any number of parameters or class type or function name. Therefore, the `advice` on line # 5 of Figure 2-3 means that the `printing` aspect should be woven `before` the `execution` of any function of any class with any return type. The match expression in the advice code (*i.e.,* `("% ...::%(...)")`) specifies that any function of any class with any return type should be treated as a join point because it uses '`...`' . In

general, the `before` (or `after`) type of advice enables specific actions to be performed before (or after) a join point. The `around` type of advice enables the execution of specific actions (the code in the body of the advice) in place of the code at the join point.

```
1.  #include<stdio.h>
2.  void A(int x, int y){
3.   printf("\nSum:  %d\n",(x+y));
4.  }
5.  void B(int x, int y){
6.   printf("\nDiff: %d\n",(x-y));
7.  }
8.  void C(int x, int y){
9.   printf("\nProduct:  %d\n",(x*y));
10. }
11. void D(int x, int y){
12.  printf("\nDivision:  %d",(x/y));
13. }
14. int main(){
15.    A(3,2);
16.    B(3,2);
17.    C(2,2);
18.    D(4,2);
19.    return 0;
20. }
```

Figure 2-2: C++ code snippet

```
1.  #ifndef _APGA_AH_
2.  #define _APGA_AH_
3.  #include <stdio.h>
4.  aspect printing{
5.      advice execution("% ...::%(...)") : before() {
6.       printf("\nFollowing Function is about to be Called\n");
7.       printf("%s",JoinPoint::signature());
8.       printf("\n");
9.      }
10. };
11.   #endif
```

Figure 2-3: AspectC++ code for printing method signature

As can be inferred from the printing aspect example (`aspect printing`), a crosscutting concern (here, printing the method signature) can be captured in a separate module, called aspect, thereby leading to untangled code. This improves software quality and maintainability by isolating the code for secondary functionality from the core computation and by reducing the size of the base program.

During the initial phase of this research, AOP was used to study the impact of separation of concerns (*e.g.,* separation of MPI-code for data distribution, communication, and synchronization from the sequential base applications) on the performance of the HPC applications. In particular, AspectC++ (the AOP language for C++) was used to weave the checkpointing and parallelization concerns (*e.g.,* MPI-code for data distribution, communication, and synchronization) into the existing C/C++ sequential applications to generate their checkpointed and parallel versions [26, 27]. The usual process of explicit parallelization requires manual insertion of library calls (*e.g.,* MPI function calls) into the existing sequential applications. By using AOP, manually inserting the library calls at multiple places can be automated while isolating the required changes (parallelization code) within aspect advice [26]. Similarly, for making the applications fault-tolerant via checkpointing, the desired code for ALC is encapsulated within aspects and woven into the existing sequential or parallel applications [27].

The initial study concluded that the separation of concerns or the usage of AspectC++ did not result in any significant degradation in the performance of generated parallel or checkpointed applications [26, 27]. The main advantages of using AOP to generate parallel applications are:

- Non-invasive reengineering of sequential applications to generate parallel applications.
- Separation of concerns, thereby leading to improved software quality, maintainability, and reusability.
- Multi-person development of HPC applications, wherein experts in parallel programming and fault-tolerance can focus on developing the

code for parallelization and checkpointing concerns and the domain-experts can focus on expressing the core computations through the sequential program.

The main challenges of using the AOP approach in the initial research and some observations are:

- Difficulty in expressing for-loops as join points [28, 29].
- Aspect weavers are available only for few languages (*e.g.,* Java, C/C++) and there is no mature and robust support for FORTRAN which is a very popular language in the HPC community.
- A large number of legacy sequential applications are non-modular and in order to apply AOP techniques on them, they should first be refactored.
- It is difficult to debug the aspect-oriented program because the woven code is not shown to the programmer [30].
- Transformed code is difficult to understand and modify [30].

Due to the aforementioned limitations associated with AOP languages, this dissertation research adopted a more general approach for transforming sequential applications into their parallel versions (and also making the sequential/parallel applications checkpointed) by using a powerful combination of generative programming approach and MDE technique.

2.2 Invasive Software Composition

ISC is a software composition approach that helps a programmer to overcome the limitations imposed by other composition techniques like AOP. ISC allows programmers

to extend the techniques as desired. ISC involves code transformation at specific change points (hooks) in components in order to adapt or extend the functionality of the components themselves. Compared to AOP, ISC, based on static metaprogramming, provides a stronger model for join points.

Reuseware [31] is an implementation of ISC used for developing semi-parallelized scientific code. Though Reuseware is not a full-fledged compiler in itself, it does works at the Abstract Syntax Tree (AST) level and falls under the category of source-to-source engines [32]. It allows the programmer to define the code blocks that should be transformed by the Reuseware engine and the action that needs to be performed by these blocks. To begin using the system, the programmer must build the composition environment by providing the grammar of the traditional language (C/C++/FORTRAN) in which the sequential program is written and the programmer must provide specifications of the composition interfaces (the extensions of the traditional language). The grammar needs to be expressed as metamodel [31, 32]. The composition interfaces are added through a *reuse language* that is provided as a grammar in EBNF. There are two basic composers that Reuseware provides for handling composition interfaces and join points: *bind* and *extend*. These composers are used to replace a composition interface or to insert fragments at specific points and can be used to build a complex composer. As noted in [32], Reuseware does not support semantic checks and does not support full-fledged pattern-matching on syntax trees.

This approach, if used without any abstraction on top of it, is low-level and invasive. The programmer must specify the MPI functions and the new variables that are to be woven in the sequential program. The programmer must be aware of all the low-level

details related to MPI-based parallelization. As noted in [32], increasing the complexity of parallelization inevitably increases the complexity of the parallelizing composers involved in the Reuseware system that involves ISC. The pattern-matching capability and the overall usage of ISC for parallelizing sequential applications still need to be made general [32] (currently it seems to be application-specific). In this research, due to the complexity involved (from the programmer's perspective) and scalability concerns, instead of adopting ISC as an intermediate component/tool, abstractions were built upon a more mature tool (source-to-source transformation engine).

2.3 Generative Programming

GP is a software development approach for modeling and developing software families such that a software system can be automatically generated from a given set of specifications and reusable components [19]. The GP-based approach adopted in this research obviates the necessity to adapt the applications to any generic interface, supports incremental integration of components, and does not require code restructuring according to any fixed guidelines. The already existing components and patterns are assembled on the basis of the high-level specifications and metadata to generate a domain-specific solution. A powerful GP-tool (source-to-source compiler that is capable of doing term-rewriting) can be used for code assembly and transformation. Several off-the-shelf source-to-source compilers are available and two such mature systems are Design Maintenance System [33, 34] and ROSE [35]. This research has used another GP-technique, template metaprogramming, to capture the patterns in data distribution, communication and synchronization. As a note, a source-to-source compiler is also

known as a program transformation engine and both terms are used interchangeably in this dissertation.

2.3.1 Template Metaprogramming

A metaprogram is a program that can manipulate or generate other programs [19]. In C++, metaprogramming can be done using templates and is based on partial-evaluation of templates at compile-time. Templates are functions or classes that are written for one or more types that are not specified at the time of writing the program. In this research, C++ template metaprogramming was used to define generic templates (type-independent) for algorithms for data distribution, synchronization, and load-balancing. These were helpful in capturing the general design pattern of parallel tasks so that they can be reused across multiple applications being developed by the framework. Templates are also helpful in optimizing code at compile-time such that the run-time overheads are saved and memory footprint is reduced. However, heavy-usage of template metaprogramming has the potential of increasing the compilation-time of applications and might lead to portability issues.

2.3.2 Program Transformation Engine

A Program Transformation Engine (PTE) is a powerful source-to-source transformation tool that works at the AST level and was used in this dissertation to overcome some of the limitations posed by the current AOP languages. A PTE uses pattern-matching and term-rewriting to carry out complex transformations. Term-rewriting is a paradigm in which rewrite rules are used to define the expected

modifications in the code structure by specifying a match-pattern and the expected effect after the rules are applied. With this approach, the source code is combined with languages and tool definitions for doing automatic analysis, enhancement, and cross-platform migration of software systems. In general, as compared to AOP, PTEs are helpful in carrying out more complex and flexible transformations because any arbitrary line of code can be specified as a join point.

The Design Maintenance System (DMS) [33, 34] is a mature and robust PTE used in this research. A major advantage in using a mature PTE like DMS is that the tool support (*e.g.,* Lexer, Parser, and Pretty Printer) is available for more than 20 languages, including FORTRAN and C/C++. Therefore, unlike the ISC technique, the programmer is free from the burden of providing the grammar of the traditional language and the grammar for creating the composition interfaces. Also, unlike the ISC technique, DMS offers powerful pattern-matching capability and can do semantic checks. The DMS is implemented in a parallel language known as PARLANSE [33, 34] and therefore, tree traversal and the transformation process are scalable to tens of thousands of files [33, 34].

As mentioned earlier, this dissertation research used DMS to transform sequential applications (base applications) into parallel and checkpointed ones (see Figure 2-4). The transformation is carried out in the DMS on the basis of the rewrite rules written in Rule Specification Language (RSL) [33, 34] and PARLANSE. The transformation rules required for this research were implemented using RSL. The external functions for complex pattern-matching required for fine-grained transformations were written using PARLANSE. These PARLANSE functions are being called from within the RSL rules and they are reusable.

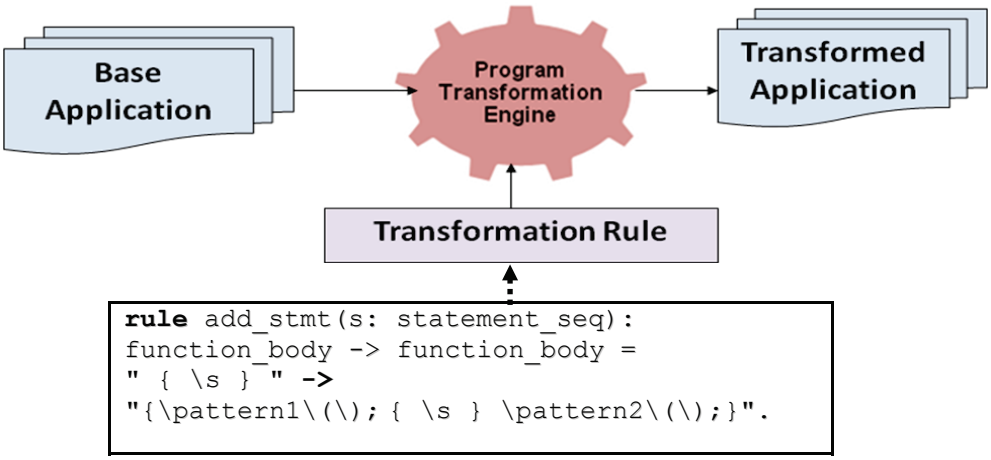

Figure 2-4: Source-to-Source transformation using a PTE

The RSL code consists of primitives (*e.g.,* pattern declaration, rules, conditions and rule sets) that are required for describing the desired source-to-source transformations. The main elements for writing RSL code are:

- **Patterns**: Used to describe the parts of the syntax tree that are of interest in the transformation process.
- **Rule**: The specification of the desired AST transformation on the basis of some conditions. It has a left-hand side, which is the source syntax expression, and a right-hand side, which is the target syntax expression.
- **Conditions**: The named boolean-valued expressions that are used to set constraints on the rules and patterns.
- **Rule Set**: Used for grouping a set of rules.
- **Default Domain**: The domain for which the rules are being written (*e.g.,* the dialect of C++ language).
- **Syntax Tree Expressions**: These are the expressions that compose trees.

The general format for specifying an RSL rule is as follows:

```
rule name_of_the_rule (parameter_list):
non-terminal -> non-terminal =
"match_pattern" -> "replacement_pattern"
if condition.
```

The words in boldface (**`rule, if`**) are the keywords. Each rule has a name defined by the programmer (`name_of_the_rule`). The pattern to be matched (`match_pattern`) is specified on the left-hand side of the rule, and the desired substitution in the syntax tree (`replacement_pattern`) is specified on the right-hand side of the rule. Constraints (`condition`), if any, are added to the rule following the keyword **`if`**. As a part of the syntax of specifying a rule, it is required that the syntax tree type (`non-terminal`) is also specified (*e.g.,* arithmetic expression, statement, and identifier). The formal parameters to the rules, patterns and conditions are optional and are represented by `parameter_list` in the general format shown above.

The transformation rule shown in Figure 2-4, transforms the body of all the functions (note that the non-terminal is `function_body`) in the program on which it is applied. The match pattern in this rule consists of a set of all the statements (`statement_seq`) inside the function body and is specified as a block '`{ \s }`'. The variable `s` is the formal parameter of the rule and '`\`' is the RSL escape character. The replacement pattern in this rule is the sequence of original statements in the function body preceded by a set of statements specified by the pattern '`pattern1\(\)`' and succeeded by a set of statements that are specified by '`\pattern2\(\)`'.

Apart from the ready-made tool support for Lexer, Parser, and Prettyprinter, DMS also provides automatically generated Rule Applier and Refiner. The Rule Applier helps in experimenting with rule sets for a single domain - it parses the input file for a domain,

applies the designated rules, and prints the results. Refiner is useful for experimenting with rules written in multiple domains. DMS also provides AST interface to support the traversal of the syntax trees that represent string-based languages. Some of the important operations on ASTs provided by DMS are finding a child node with a specific property, finding a parent node with a specific property, getting the Nth child node from a parent node, *etc*.

For all the test cases used in this research, the performance of the PTE-generated code was similar to the manually written code. The generated code is the same as the manually written code and the PTE does not introduce any artifacts. However, it is difficult to learn and use the PTE (DMS in particular). The rewrite rules are difficult to write and the ambiguity errors during the transformation process are hard to understand and resolve. Also, the rewrite rules are very domain-specific, grammar-specific and PTE-specific. A rule written in a particular dialect of C++ (*e.g.,* Microsoft's Visual6 C++) might not work with another dialect (*e.g.,* GNU C++). If, for example, the DMS is replaced by another PTE, the already written RSL-rules would become useless. Certain search patterns are too complex to express through RSL code and therefore external functions should be called by the RSL rules for pattern-matching and syntax tree manipulation. Such external functions are written in PARLANSE which is a functional language that is difficult to learn due to lack of extensive documentation and the need to manually handle the syntax tree traversal and transformation. Lastly, the DMS is a costly commercial tool and might not be available to all the developers participating in the application development lifecycle. These limitations were the motivating factors behind exploring techniques for abstracting out the code transformation process from the programmer. The intent was to

benefit from the power and robustness of a PTE while reducing the accidental complexities associated with its usage.

2.4 Domain-Specific Language

DSLs are specialized languages written for a particular application-domain [20]. They raise the level of abstraction of programming and enable the domain-experts to work in a language closer to their problem domain. As compared to the General-Purpose Languages (GPLs), DSLs are more expressive in a given domain but have limited features and applicability. Like any GPL, a DSL has a concrete syntax, abstract syntax, and well defined semantics. Because the DSLs are more specialized and expressive than other GPLs, they are easy to learn and use. The usage of a DSL increases productivity and decreases software development time and cost [20]. An excellent example of a DSL is Structured Query Language (SQL) which is related to the database domain.

The first step in developing a DSL is analyzing the domain for which it is being designed. During the domain analysis phase of developing the DSLs required for this dissertation, technical literature and existing implementations were surveyed to obtain an overview of the terminologies and concepts related to the domains of concern. Commonly used terms and their relationships were used to develop the domain lexicons. Feature-Oriented Domain Analysis (FODA) [19] was used for further domain analysis. FODA is often used to develop generic domain products by employing abstraction and refinement. The specific applications from a domain are analyzed and a layer of abstraction is added to hide the differences between the applications. The generic product can then be refined to generate a specific application [19]. A feature model represents the

commonalities and differences between various features of an application. As advocated in [36], only the necessary and relevant features were modeled. The next steps for developing the DSL are domain design and implementation. Already existing notations were adopted for the DSLs developed for this research and new terms and jargon were avoided. The concepts and constructs of the DSLs developed in this research have some resemblance to the AOP language concept (the join-point model [18]) and constructs (*e.g., before*, *after*, and *around*). The implementation of the DSLs was done using the MDE technique.

2.5 Model-Driven Engineering

MDE is a software engineering paradigm that involves abstraction of real-world entities or concepts as models. With the help of a translator or an interpreter, the actual code can be automatically generated from the models. Hence, MDE raises the level of abstraction of programming in high-level languages and helps in expressing domain-specific concepts efficiently. The MDE technique was used in this research for developing the DSLs that are required for obtaining the specifications for parallelization and checkpointing from the programmer. The DSLs (and hence the MDE paradigm) were helpful in making the framework flexible and extensible so that multiple traditional languages and parallel programming paradigms can be supported with minimum effort [37]. Some of the MDE-concepts are defined as follows:

- **Model:** A model is a representation of a system and can be of three types - a terminal model, a metamodel or a metametamodel.
- **Terminal Model:** A model whose reference model is a metamodel.

- **MetaModel:** A model that defines a language for expressing other models. It describes different contained model elements and the relationship between them. It conforms to a metametamodel and Meta-Object Facility (MOF) is an example of a metametamodel. The MOF metametamodel is self-defined.
- **MegaModel:** A megamodel is a model that records the global information on tools, services and other models.

The MDE platform used in this research is called Atlas Model Management Architecture (AMMA) [38] and is implemented on top of the Eclipse/EMF framework. AMMA provides a set of facilities for processing models and consists of the following main blocks [39]

- **Kernel MetaMetaModel (KM3):** It is an implementation-independent, textual domain-specific language for defining the abstract syntax of the DSLs in the form of metamodels [40]. KM3 resembles the Ecore terminology [41] and uses concepts like package, class, attribute, reference, and primitive data type.
- **Textual Concrete Syntax (TCS):** TCS [42] is itself a DSL and can be used to specify the textual concrete syntax of other DSLs by attaching syntactic information to metamodels. TCS-specifications are used to automatically generate tools for model-to-text (by generating ANTLR grammar) and text-to-model transformations (by using a Java-based extractor) [42].
- **Atlas Transformation Language (ATL):** A model transformation language transforms a set of source models into a set of target models on the basis of the defined rules [43]. ATL has its abstract syntax defined as a metamodel and every ATL transformation is itself considered as a model. The language consists of rules

and expressions (based on OCL [44]). Each rule consists of a source pattern on its left-hand side and a target pattern on the right-hand side. The source pattern consists of model element types from the source metamodels and the boolean expressions. The target pattern consists of the model element types from the target metamodel. A set of binding is attached to each rule for specifying the way in which the properties of the target elements should be initialized.

- **Atlas MegaModel Management (AM3):** Provides support for global resource management in a model-engineering environment. The main features of AM3 are management of megamodels, management of various relations between artifacts (*e.g.,* models, metamodels, transformations, and semantic correspondences), sharing and exchanging of megamodel elements, and user interfaces for viewing megamodel elements [45, 46]. Because the AM3 supported megamodels allow the manipulation of other resources such as XML documents, database tables or flat files as well, the notion of artifacts is used in general.

2.6 Checkpointing

Checkpointing is a mechanism by which an application is made resilient to failures by periodically saving its state to the secondary storage medium. Scientific applications that take an enormous amount of time to execute (*e.g.,* simulation for protein structure prediction [47] or climate modeling [48]) and are run in distributed, dynamic and heterogeneous environments, like a grid, can benefit considerably from checkpointing. In case of failures or changes in the availability of underlying resources, instead of restarting the application from the beginning, the programmer can restart the application from the

latest checkpoint. This is achieved by recreating the pre-failure application state from the saved data on the disk. Checkpointing is also an essential component for writing self-healing applications. These applications can monitor their own state, detect faults, and recover from the faults automatically. Checkpointing or a similar mechanism (*e.g.,* logging) is required to recover from the fault and continue execution without having to restart the whole application.

Writing and reading the application state are the major steps involved in checkpointing. Through the rest of this dissertation, these steps are referred to as Checkpointing and Restart (CaR). The main types of checkpointing techniques, depending upon the level of transparency, are: hardware-level [49], system-level [50], user-level [51], application-level [52, 53, 54], and hybrid approaches [55].

- In the *hardware-level* checkpointing, specialized hardware (*e.g.,* redundant arrays of inexpensive disks, custom-designed directory controller, and cache memory) can be integrated into the processors for saving the state of the application.
- The *system-level* checkpointing is performed external to the application with the support of the operating system and involves periodically saving the execution state of the entire application. Typically, this requires changes to the operating system's kernel and the entire process state is saved since the operating system does not have knowledge about the application semantics.
- The *user-level* checkpointing process is often accomplished by linking the checkpointing libraries to the application code. The programmer is free from the burden of making any changes to the code and no additional code is required to be

installed in the kernel as compared to the system-level checkpointing. This approach is usually architecture-dependent.

- In the *Application-Level Checkpointing (ALC)* an application is made reliable by inserting the fault-tolerance mechanism directly into it. Only the critical variables and data structures are saved to the disk during ALC.
- A *Hybrid Approach* is a combination of multiple checkpointing techniques. A hybrid of system-level checkpointing and ALC is presented in [55]. The authors claim that this combination results in higher reliability in real-time systems.

Although ALC requires more end-user involvement than any other form of checkpointing, it has several advantages [56]. As compared to other types of checkpointing techniques, ALC involves lesser storage space (core-dumps are taken in system-level checkpointing), offers the end-user more control for selective checkpointing and is useful for writing portable applications for different operating systems. As mentioned in [56], the ALC schemes are language-independent "provided that the base language constructs are present" [56].

One major problem with the current techniques for ALC is that the current techniques require invasive reengineering of existing applications to insert the checkpointing code (typically done by inserting macros in the source code) and thus make software maintenance challenging. If the application code is large, and the number of critical variables is huge, there may be multiple places where the end-user must make changes in the existing application to make it fault-tolerant via checkpointing. Because ALC involves extra read and write operations, the checkpointed version of the application might take substantially longer time to run than the non-checkpointed one. When

performance is more critical than fault-tolerance, the stakeholder might want to have the facility to turn-off the checkpointing feature. For the convenience of code maintenance and evolution, it is important to avoid creating multiple versions of the application (with and without checkpointing). Also, the solution space for ALC is constantly evolving. Many checkpointing libraries and techniques exist and each has some special merits over the others [50, 56]. With the emergence of many-core and multi-core architectures, more solutions for fault-tolerance are expected to emerge. Given such a widespread and an evolving solution space, the end-users should not be forced to reengineer their application to switch from one solution to another. Due to all these reasons, it is desirable that the existing application should not undergo any invasive reengineering in order to become fault-tolerant and the CaR mechanism (to enable ALC) should exist as a pluggable feature.

In this dissertation, the checkpointing mechanism is ALC-centric. This work is relevant for both uniprocessor and multiprocessor systems. At a coarse-grain level, it can be said that the ALC-approach developed in this research is selective (core-dump of the processor's state is not taken), periodic (checkpoints are always taken at a particular frequency), and static (because the checkpoints are known before the program is run). This approach applies to checkpointing both sequential and parallel programs. When checkpointing parallel applications with this approach, depending upon the end-user's choice, checkpointing can be centralized (only one processor initiates the checkpoint) or distributed (each processor participates in the checkpointing process). Because while taking centralized checkpoints with this approach, it is important that the processors are in a synchronized state, this approach is a coordinated one. However, synchronizing the

processors is not a part of the approach. It is the end-user's responsibility to manually ensure this.

This research's focus is to raise the level of abstraction of ALC so that the end-user is not responsible for manually reengineering the existing application to insert the checkpointing code. Instead of writing the optimized ALC code by hand or inserting any library calls in the code (which could lead to code tangling), the end-user provides the CaR-specifications (what should be checkpointed and where, along with the frequency of checkpointing) using the DSL developed in this research. The necessary code is then generated and inserted into the existing application using a set of domain-specific optimizations (*i.e.,* transformations) [19]. This approach solves the problems related to versioning and maintenance (described above) and allows the end-users to take advantage of the latest tools and techniques for ALC. Other advantages of this high-level approach include: enhanced code reuse, absence of code restructuring, and highly comprehensible/readable code for the CaR mechanism. This research also provides the facility to checkpoint code at arbitrary points in the application. The applications that were checkpointed through the technique demonstrated in this research produce results with the same accuracy and precision as the non-checkpointed code or the manually checkpointed code. The performance of the application checkpointed by this technique is comparable to the manually checkpointed version of the application.

2.7 Related High-Level Parallel Programming Approaches

There have been several research efforts in the past to raise the level of abstraction of parallel programming. Various libraries, toolkits, languages, and language

extensions have been developed and a detailed discussion on these can be found in [57, 58, 59, 60, 61, 62, and 63]. Most of these techniques demonstrate the significance of abstractions and component-based HPC application development. It has been noted in these papers that the reasons behind the failure of some high-level approaches from the past are lack of flexibility, performance, and extensibility [57, 58, and 59]. In other words, in order to be widely accepted and adopted, any high-level approach should guarantee a reasonable amount of performance while providing the flexibility to modify the generated application. It should also be extensible enough to support multiple platforms.

Some of the efforts for raising the level of abstraction (those that use design patterns and templates [60], Invasive Software Composition (ISC) [30, 32, and 61], Aspect-Oriented Programming (AOP) [26, 27, 28, 29, and 62] and skeleton-oriented frameworks [63]) have successfully demonstrated the advantages of keeping parallel and sequential concerns separate in order to reduce code complexity and thereby augmenting code reuse and making the process of code maintenance easy. The lessons learned from these approaches are reflected in this dissertation research. A discussion on some of the techniques that are complementary to FraSPA is presented in the following subsections.

2.7.1 **New Parallel Programming Languages**

Partitioned Global Address Space (PGAS) languages like UPC [64], Co-Array Fortran [65], X10 [11], Chapel [66] and Titanium [67] do have advantages over MPI, for instance, they offer better performance for fine-grained communication, yet they are still evolving. In order to use these languages (or language extensions), legacy applications

must be reengineered invasively and the application developer must be familiar with these new programming paradigms. Some other parallel programming languages that fall under the category of fine-grained mechanisms for parallel computations are Orca [12], SISAL [13], and Fortress [14].

As noted in [15], the disadvantage of new parallel languages for implicit parallelization, like SISAL, is that the programmer might have limited flexibility to experiment with different algorithm-design options. Debugging for performance is also difficult because it is unclear which code construct might be contributing to the loss in performance. That is because the compiler is in control of parallelization [15]. Legacy code, and demand for Fortran/C-based languages make these new languages impractical options.

One concern with the new programming languages is the steep learning curve for the programmers. Another concern is that the programmers have to rewrite the applications in the new language, do the required testing and debugging, thus making the process time consuming and expensive. Many applications that are already in production have considerable amount of time and money already invested in them. It is not a trivial task to rewrite them from scratch. Instead, libraries for parallelization (*e.g.,* MPI) help programmers to take advantage of the existing code. Therefore, a layer of abstraction built on top of such popular libraries has a potential of reducing the effort involved in parallelization without involving any steep learning curve. A discussion of some of the techniques that have the objectives closely related to this research (MPI-based, high-level, generative programming-based, applicable to a wide range of application domains, and performance-oriented) is provided below.

CHARM++ [68] is an explicit parallel language and adaptive runtime system that is based upon the object-oriented programming paradigm having its roots in C++. The execution model of CHARM++ is asynchronous and message-driven (or event-driven). The programmer must decompose the computation into small virtual message-driven processes called as chares. The data mapping to processors, fault-tolerance and load-balancing is done by the system. The programmer must also write an interface file to provide the methods of the chare object that can be invoked by other chare objects. The language can be used for writing Multiple-Instruction Multiple Data (MIMD) parallel programs for both shared memory and distributed memory applications and handles the portability issues really well. This approach provides separation of sequential and parallel concerns and is best-suited for applications meant to be written from scratch. A steep learning curve is associated with the usage of CHARM++. FraSPA provides support for most of the features for parallelization that are supported by CHARM++ including fault-tolerance, and is best suited for reengineering legacy sequential applications.

Sequoia [69] is a programming language designed for the development of the memory-hierarchy aware portable parallel programs. The generic algorithmic expression and machine-specific optimization are kept strictly separate to enable portability without compromising on performance [69]. In Sequoia, the application developer abstractly describes the hierarchies of tasks and then maps these hierarchies to the memory system of a target machine. This process entails a complete rewrite of the existing application to take advantage of the abstraction provided by Sequoia. Unlike in Sequoia, FraSPA does not necessitate the rewrite of the existing application for parallelizing it. As in Sequoia,

the FraSPA provides the separation of machine-specific optimization and generic code constructs.

High Performance FORTRAN (HPF) [70] is a high-level language extension of FORTRAN 90 used to annotate the FORTRAN code with directives for data decomposition, specifying data parallel operations and mapping data to the processors. The high-level language developed in this research has the potential to support parallelization of languages written in multiple programming languages including FORTRAN.

2.7.2 **Pattern-Based Approaches**

Design Patterns and Distributed Process (DPnDP) [60] and MPI Advanced Pattern-Based Parallel Programming System (MAP_3S) [71] are pattern-based systems for developing parallel applications. These systems are extensible, flexible and demonstrate the advantages of keeping the sequential and parallel concerns separate in an application (and thereby reducing code complexity) while promoting code reuse and correctness. The DPnDP system generates the code skeleton depending upon the parameters (*e.g.,* master-slave pattern and the number of processors) specified by the programmer. The programmer is then required to edit the generated files that contain code skeletons to insert the sequential entry procedures into the skeleton. The MAP_3S system demonstrates that generative pattern systems can be successfully implemented using the combination of MPI and C. The system uses customization and tuning parameters provided by the programmer to develop efficient parallel code templates. The programmer is also required to provide macros for packing the processing elements into MPI packets.

However, low-level specifications, like the maximum size of packet that can be transmitted between the processors or the timing for synchronization, place the burden on the programmer to understand the limitations of the MPI. Compared to DPnDP and MAP_3S, FraSPA does not involve any intrusive reengineering of the existing code and the programmer is not required to specify any low-level parameters. Also, DPnDP and MAP_3S systems seem to be better suited for applications that are meant to be written from scratch. FraSPA is better suited for reengineering existing sequential applications in order to parallelize them.

Tracs [72], developed at the University of Pisa, provides design pattern components from which an actual application can be built. Tracs requires that all design patterns be expressed graphically. However, this is its major limitation because not all the design patterns can be expressed graphically (*e.g.,* divide and conquer).

MPIBuddy [59], a portable design pattern based system for parallel programming implemented in Java, provides a level of abstraction above MPI and is an extensible system. MPIBuddy has a graphically rich front-end written in Java and uses Java Native Interface (JNI) methods to use C and MPI code. The interface programming offers the advantages of both the efficient low-level code (C and MPI) and easy graphical development through Java. This approach provides a skeleton that should be manually fleshed out with application-specific code. The user should be aware of parallel programming constructs and APIs (*e.g.,* MPI APIs). In contrast, the users of the FraSPA are not required to know the intricacies of parallel programming except identifying concurrency and selecting the right type of functionality (*e.g.,* distribute data, gather data).

2.7.3 **Domain-Specific Approaches**

Spiral [7] is a domain-specific library generation system that generates HPC code for some domains (*e.g.,* linear transformations) using the high-level specifications. It supports a wide range of platforms for complete automation of implementing and optimizing libraries. The basic idea behind Spiral is to capture the algorithm at a high-level and use rewrite systems to generate the executable. Spiral uses a feedback mechanism for exploring the solution space (set of candidate solutions) in order to pick the highly optimized solution for a particular platform. Unlike Spiral that generates optimized libraries, this research aims to generate complete parallel applications from existing sequential applications. Architecture-specific optimizations for any particular domain are beyond the current scope of this research.

Catanzaro *et al.* [73] are developing a set of DSLs for different application-domains (a different DSL for each application domain under study) to simplify the process of developing applications for heterogeneous architectures. Though their research goal (generating parallel code from high-level languages for improving end-user productivity) seems to be similar to that of FraSPA, there are major differences in the details associated with the two efforts. FraSPA is useful for synthesizing optimized parallel applications in a non-invasive and domain-neutral manner, *i.e.* it can parallelize sequential applications from diverse domains. A single DSL has been developed in this research to capture the specifications of parallel tasks per se (*e.g.,* reduce data, gather data, and distribute data) and the end-users are not required to specify their core-computations in any particular fashion (or in conformance to any standard interface). In contrast, the set of DSLs from Catanzaro *et al.* capture the domain-specific computations

(for mesh-based PDE, physics library PhysBAM, and machine-learning), promise to deliver optimized parallel solutions, and seem to be suitable for applications that are to be written from scratch. Their framework seems to be still under development and they have yet to develop mechanisms that provide communication and synchronization with low overhead [73]. The current scope of FraSPA is limited to providing the facility for parallel code-generation for homogeneous architectures, and the optimal solutions are generated through the usage of design patterns and templates.

Google's "MapReduce is a programming model and an associated implementation for processing and generating large data sets. Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system." [74] MapReduce is an abstraction that is helpful in expressing simple computations without getting into the details of parallelization, fault-tolerance, data distribution, and load-balancing. It seems to separate the computation concerns from the parallelization concerns to some extent but there is not much information publically available on this. In FraSPA, the end-user if not required to write the computations according to a particular model.

Similar in functionality to Google's MapReduce, an open-source project called as Hadoop also provides a MapReduce Framework [75] for performing computations in parallel. Hadoop also provides a distributed file system for storing data on compute nodes

and node failures are handled by the framework itself. MapReduce has two phases of computation – a map phase and a reduce phase. In the map phase, the input data set is split into multiple fragments and distributed to compute nodes by the framework. A key-value pair is provided as input to each map task which produces an intermediate key-value pair as output by invoking the user-defined map function. The intermediate results are sorted and each key can have multiple results associated with it. These intermediate results are consumed by the reduce task that invokes the user-defined reduce function for generating an output of key-value pairs.

2.7.4 Library-Based Approaches

Application-specific libraries and toolkits for parallelization (*e.g.,* PBLAS [76], POOMA [77], PETSc [78] and BlockSolve [79]) provide high-level and fine-grained mechanism for parallelization. These libraries provide a very application-specific parallel solution. With toolkits or library, the user writes the main body of the application and inserts calls to reusable routines (that the toolkits and the libraries provide). However, with FraSPA, the end-user is not supposed to make any changes to the existing sequential application and has the flexibility to generate applications from diverse domains.

2.7.5 Other Related Work

Chamberlain *et al.* have shown an approach for designing and developing an environment for authoring and deploying applications on hybrid systems [80]. Their solution involves the use of a coordination language to specify dataflow style interconnections between compute blocks, and native languages and tool sets for the

development of the compute blocks themselves. The application performance is evaluated early in the design cycle. Due to the environment support, the compute blocks are mapped to the computational resources in a semi-automated way. The environment also supports block-to-block communication both within and between computational resources [80]. With this approach the user is responsible for writing the compute blocks for all the different architectures participating in the hybrid system. This is an excellent approach for writing applications from scratch. The experts in the languages and tools for each type of architecture can write their implementations individually and can later integrate the different implementations via the glue code provided by the coordination language. FraSPA is different from this approach because it automatically synthesizes an MPI application from an existing sequential application on the basis of the high-level specifications provided by the end-user.

Roychoudhury *et al.* [37, 81] have demonstrated a technique for constructing aspect-weavers for general-purpose programming languages by combining model-driven engineering with a program transformation system. In their technique, the aspect-specifications are captured in an abstract manner such that there is no dependency on any one particular program transformation system. Their framework for constructing aspect-weavers has influenced the design-decisions of FraSPA in a manner that the high-level specifications for parallelization and checkpointing are decoupled from the low-level implementation details.

2.8 Related Approaches for Fault-Tolerance Through Checkpointing

Bronevetsky *et al.* [52, 53, and 54] have proposed a preprocessor-based approach for ALC. Their work is relevant for both shared memory and distributed memory architectures and their approach consists of two components: a pre-processor, and a checkpointing library. With their approach, the programmer invasively changes the existing application to insert the calls to a predefined function for checkpointing, at the points in the program where checkpointing is desired. An optimized approach to automated ALC is presented in [54], which is helpful for asynchronously checkpointing an application.

Ramkumar *et al.* [82] have used a source-to-source compilation technique for creating portable checkpoints. In their approach too, the end-user has to instrument the existing code by renaming functions and by inserting the call to the checkpointing library function. The frequency of checkpointing is controlled using a timer that triggers checkpointing. The state of the program is stored on stacks and this approach doubles the memory requirement for running an application. In case the DRAM cannot hold the data on the stack, then the stack is mapped to a local disk and thus extra checkpointing overheads are introduced.

Jiang *et al.* [83] proposed an ALC technique for shared-memory architectures which they call MigThread. This technique consists of a LEX-based preprocessor and a runtime support module. The preprocessor scans the code and inserts the thread migration primitives, renames the functions and variables and inserts other code required for thread migration [83]. In this technique, parts of computation are assigned to different threads,

the computation is paused, the state of the threads (process, computation, communication) is migrated to a different node, and the computation is resumed.

In [84], Czarnul *et al.* have proposed a user-guided approach for inserting calls to their checkpointing library, either through a dedicated master processor or collectively by all the processors, and call it PARUG. This approach offers the flexibility of selective checkpointing to the end-user but is invasive.

The major differences between the checkpointing approach developed in this research and other related work are the non-invasive reengineering of existing applications, separation of the checkpointing concern from the existing application, and the readability/comprehensibility of the generated code. However, the onus is on the end-user to identify the places in the code where checkpointing is required and to specify the checkpointing-frequency. As compared to Bronevetsky *et al*.'s approach, the research presented in this dissertation is non-invasive but semi-automatic and the end-user is responsible for ensuring that the processors are in a consistent state before taking the checkpoint. As compared to Ramkumar *et al*'s approach, the research presented in this dissertation is at a very high-level of abstraction, gives control to the end-user to select the critical program variables to checkpoint and to select the frequency of checkpointing. As compared to MigThread, the research presented in this paper is relevant for different types of architectures and the transformed code is more comprehensible to the end-user because the original structure is maintained as is with the exception of checkpointing code inserted at the specified places in the application. The work done in this dissertation can be extended to support non-invasive ALC of applications written in several base languages, including FORTRAN [37, 81].

2.9 General Discussion

Out of all the approaches presented in Sections 2.7 and 2.8, at the time of writing this dissertation only FraSPA, CHARM++, and MapReduce, provide a high-level approach for developing fault-tolerant parallel applications. Amongst the three, FraSPA is best-suited for the scenarios in which the legacy applications already exist and the end-users intend to transform these applications to parallelize them or make them fault-tolerant via checkpointing. Both CHARM++ and MapReduce are suitable for the scenarios in which the applications are being written from scratch. While both FraSPA and CHARM++ can be used to develop parallel and fault-tolerant applications from diverse domains, MapReduce has limited functionality because not all the applications can be solved by the MapReduce algorithm. MapReduce is best suited for data-parallel applications which process vast amounts of data. While FraSPA and CHARM++ support the notion of separation of concerns, in case of MapReduce, the functionality for map and reduce operations could be intertwined with the core computations – though the parallelization is hidden from the end-user, but the end-user is still required to implement map and reduce functions in their applications in conformance to standard interfaces.

The high-level approaches that are closely related to this dissertation are loosely classified in Figure 2-5. There are five criteria according to which the classification has been done in Figure 2-5 and they are:

- Approaches that are language-based - either application domain-specific (*e.g.,* Spiral, MapReduce and Catanzaro *et al*.) or application domain-neutral (*e.g.,* UPC, Orca, CHARM++, and FraSPA).

- Design pattern-based approaches (*e.g.,* DPnDP, MAP_3S, Tracs, MPIBuddy, and FraSPA).
- Approaches that support separation of concerns (*e.g.,* DPnDP, MAP_3S, CHARM++, and FraSPA).
- Approaches that support generation of applications for heterogeneous architectures (*e.g.,* Spiral, Catanzaro *et al*., and Chamberlain *et al*.).
- Approaches that provide support for fault-tolerance (*e.g.,* MapReduce, CHARM++, and FraSPA).

Only MapReduce, Spiral and Catanzaro *et al.*'s work can be classified as application domain-specific. Most of the other approaches are application domain-neutral. Given the current scope of FraSPA, it meets four out of five classification-criteria and has the potential of being extended to meet the fifth criterion as well - which is, providing support for synthesizing applications for heterogeneous architectures. The DSLs (Hi-PaL and DALC) developed as part of FraSPA are application domain-neutral. FraSPA clearly supports separation of concern and fault-tolerance via checkpointing. It uses design-templates that are codified design patterns for data-distribution, load-balancing and synchronization for distributed memory HPC platforms.

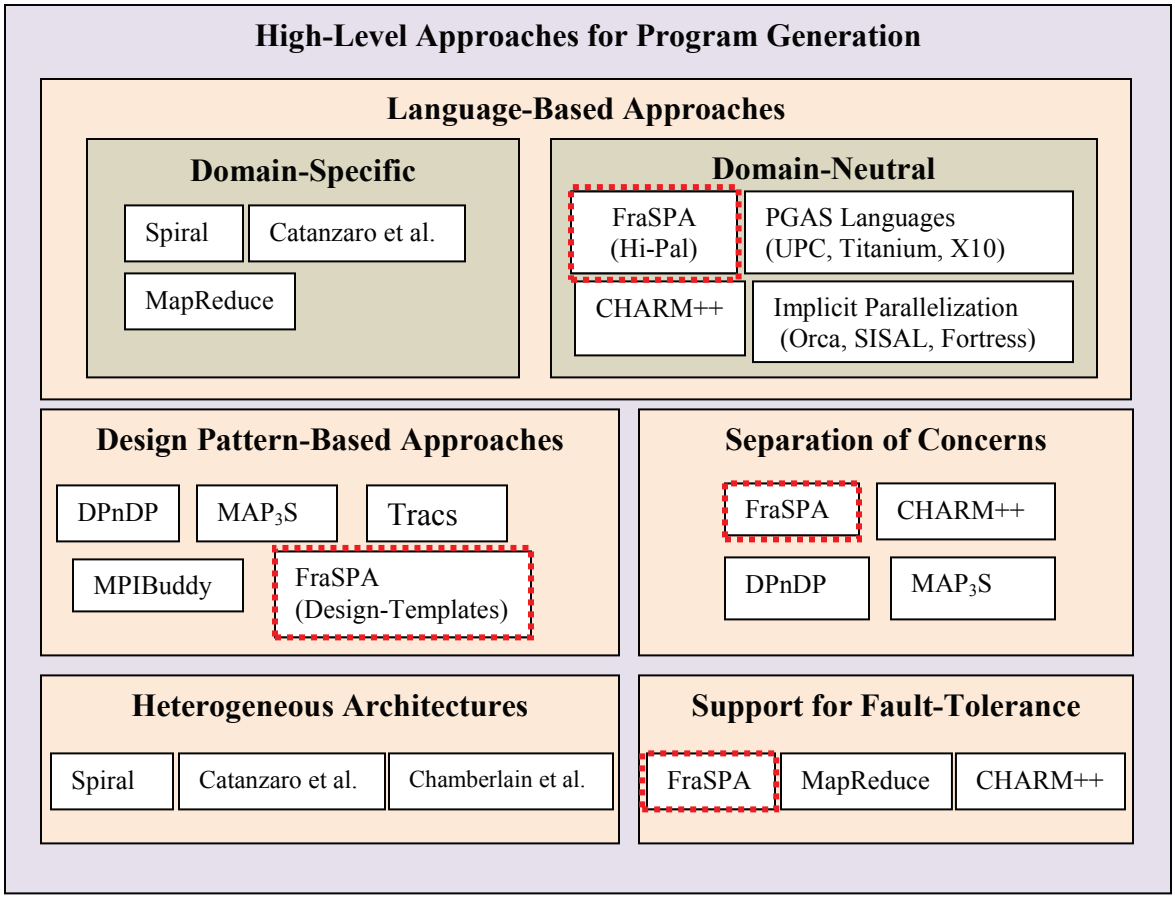

Figure 2-5: High-Level approaches for parallel program generation

CHAPTER 3

DESIGN AND IMPLEMENTATION OF FRAMEWORK

As described in Chapter 1, the goal of this research was to raise the level of abstraction of generating performance-oriented, checkpointed parallel applications from existing sequential applications. This would reduce the effort and complexities associated with developing HPC applications for distributed memory platforms. As explained in Chapter 2, during the initial phase of this research, multiple approaches were explored to achieve this goal. It was found that no single approach or tool can address all the requirements alone. Only through a combination of modern software engineering tools and techniques can the major challenges associated with achieving the dissertation's goal be solved. As evident from the related work sections in Chapter 2, developing high-level parallel programming environments is an actively researched area. The lessons learned from the success and limitations of the complementary approaches have guided the development of a framework, named FraSPA, in this research. The main limitations of the complementary approaches in the light of the goal of FraSPA are their application domain-specific nature and the requirement to write the applications from scratch. The main lesson learned from the complementary approaches is that a widely accepted framework should provide support for separation of concerns, fault-tolerance mechanism, flexibility and extensibility. The lessons learned are embodied in the form of FraSPA which is an application domain-neutral, component-based framework for generating

checkpointed parallel applications from existing sequential applications and high-level specifications. FraSPA supports separation of concerns and does not entail any manual reengineering of existing applications to achieve the objectives of parallelization or fault-tolerance via checkpointing.

Currently the scope of FraSPA is limited to the generation of checkpointed parallel applications for homogeneous HPC platforms. To obtain the bigger goal of extending the framework for generating checkpointed parallel applications for heterogeneous architectures and multiple programming languages (C/C++/FORTRAN), it was required that the framework be extensible and flexible. If the framework is extensible, the support for generating applications for heterogeneous architectures can be added in the future without making any changes to the existing code. Currently, FraSPA supports the synthesis of applications that are written in C/C++ as base languages. However, because of its extensible nature, it can be extended to support generation of applications written in FORTRAN as well. Through their work on generic aspect weavers, Roychoudhury *et al.* [37, 81] have demonstrated a technique for achieving extensibility in a framework. Because similar design principles are adopted for the development of FraSPA, it can be inferred that FraSPA has the desired property of extensibility to support other languages.

FraSPA is flexible and by using various components in isolation or together, it can support the composition of parallel and checkpointed applications. For example, the API for distributing the data amongst various processors can be used in conjunction with the API for gathering the results or reducing the results without any restrictions. This approach gives the end-user the flexibility to experiment with multiple communication

patterns and algorithms. However, the onus is on the end-user to correctly choose the parallel operations to be inserted at a particular place in the base application.

The terms framework, extensibility, and flexibility as used in this dissertation are defined below:

- Framework: A framework is a software system that abstracts selected common functionality in the form of generic code which can be specialized according to the end-user's needs. In essence, it is a consolidation of various generic code components (*e.g.,* rule generator, templates for communication patterns, and Ant Scripts for invoking the PTE) for achieving the desired abstraction. The flow of control (which component to use and when on the basis of the high-level specifications) is automated and is not in the control of the user of the framework.
- Extensibility: The property of the framework which allows the user to add extra functionality at a later stage is called extensibility. In context of FraSPA, extensibility is desired so that the support for multiple programming models (*e.g.,* OpenMP, and OpenCL) and languages can be provided in future.
- Flexibility: The property of the framework that allows the user to compose their own design patterns from the existing components by assembling them in any order is called flexibility. In the context of FraSPA, this means that there is no restriction in the order of specifying the API for different parallel operations. Also, there is no restriction imposed by FraSPA on the combination of various parallel operations.

The rest of this chapter describes the overall approach, architecture, design, and implementation of FraSPA. An overview of the overall approach for synthesizing

checkpointed parallel applications is presented in Section 3.1. Design details and the high-level architecture of FraSPA are presented in Section 3.2 and the implementation of FraSPA is described in Section 3.3. A detailed description of the main components, concepts, and tools related to the design and implementation of FraSPA has already been presented in Chapter 2. A summary of the chapter is presented in Section 3.4. The terms "Source-to-Source Compiler" and "Program Transformation Engine" are used interchangeably in this chapter.

3.1 Overview of the Approach

This research involves source-to-source transformation by the means of high-level specifications provided by the end-user in the form of DSL code. As shown in Figure 3-1, the DSL code is parsed by the Rule Generator to generate intermediate code. The intermediate code is a set of rules (Generated Rules in Figure 3-1) that the Source-to-Source Compiler (SSC) can comprehend. These rules contain the precise information about the modifications desired by the end-user, and the place in the Existing Code where these modifications should take effect. By applying the Generated Rules and other optional code components (*e.g.*, Design-Templates) the SSC transforms the Existing Code into Transformed Code. In this research, both the Existing Code and Transformed Code have the same base language (C/C++) but even if they were in different languages, this approach would still work [33, 34].

The complete work-flow - from the first step, which is the parsing of the input DSL code, to the last step, in which the transformed code is generated - is part of FraSPA. The set of DSLs developed in this research serve as the interface between the

end-user and FraSPA. The end-users analyze their existing applications and can express the transformations that they desire by the means of the DSL. The other steps responsible for transformations are like a "black-box" to the end-users. Therefore, the approach used in this research can be used to raise the level of abstraction of source-to-source transformations. Through this approach, the end-users (including domain experts) can express the specifications of what they intend to do (explicit parallelization and checkpointing for fault-tolerance), and are freed from the burden of how their intentions are materialized because low-level programming and source-to-source transformation details are hidden from them.

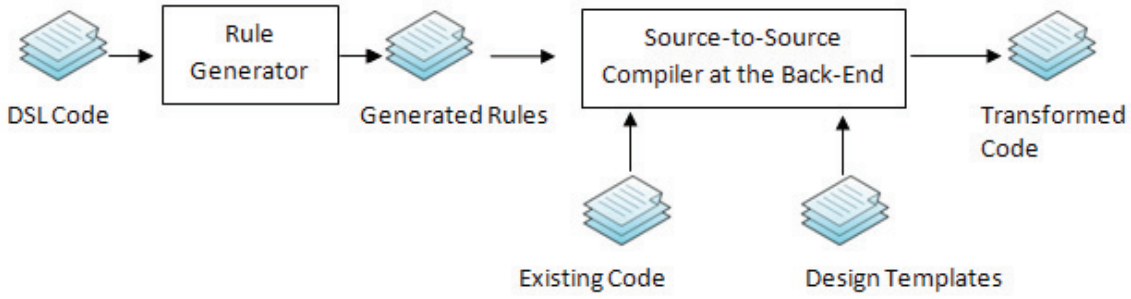

Figure 3-1- High-Level idea behind the working of FraSPA

Approaches similar to the one used in this research, are being adopted by other researchers [7, 73] who realize that the DSL-route has the potential to reduce both the time-to-solutions and the complexities associated with HPC application development. Most of the challenges identified in Chapter 1, can be mitigated with the approach presented in Figure 3-1 because this high-level approach does not entail invasive manual-reengineering of existing applications, is platform-independent, and also base-language-independent. Unlike the approaches in [7 and 73], the approach adopted in this research is application-domain neutral.

Before reaching the current DSL-based design of FraSPA, direct usage of source-to-source transformations techniques for generating checkpointed parallel applications was also explored. AOP has the potential for doing the desired code-weaving and transformation. However, due to the limitations of the current implementations of AOP languages (described in Chapter 2), a robust PTE was chosen in this research to carry out the required transformations. Because the PTE is capable of doing non-invasive transformation by itself, it could have also been used directly to transform the existing application into a parallel or a checkpointed one. This would have obviated the extra effort spent in developing the DSLs. However, PTEs are complex and difficult to learn and use.

The PTE used in this research, described in Chapter 2, is DMS [33, 34]. The end-user has to climb a steep learning curve in order to use DMS. It is also required to develop an understanding of the base language grammar and the various tools available through DMS. Of the many features and tools associated with DMS, the end-users should at least have an understanding of RSL, PARLANSE, and the usage of AST API provided with DMS in order to begin using the system for source-to-source transformations. The debugging facility provided in DMS is very basic and the error messages are often difficult to understand (*e.g.,* foreign exceptions and ambiguity errors at runtime). Some search patterns are difficult to specify directly in RSL rules. Hence, external patterns in PARLANSE should be written and called from within the RSL rules.

To leverage the powerful source-to-source transformation capabilities of DMS (in-built Lexer, Parser, Rule Analyzer and Pretty Printer), while avoiding the complexities associated with its usage, an extra layer of abstraction in the form of DSLs

in the front-end was absolutely necessary. In summary, using DSLs in this research not only helps in non-invasive reengineering of existing applications, platform-independence, and base-language-independence, but also mitigates the complexities involved with source-to-source transformation techniques.

To fulfill the requirements in the current and future scope of this research, the framework had to be extensible and flexible. In case, a concept or a feature is missing from the framework, it should be possible to later add that to the framework without modifying the existing code. FraSPA currently supports the abstractions for some commonly used MPI primitives. The support for various parallel operations (*viz.* distribute, gather, and reduce) was incrementally added to the framework. If support for additional parallel operations is required in future, the same can be added to FraSPA without modifying the code for the supported parallel operations.

The components in the FraSPA design are decoupled from each other so a component in a particular layer can be replaced with another without impacting the components in other layers. For example, if DMS in the back-end is replaced with another PTE, it will not impact the implementation of the DSL in the front-end. Likewise, if the implementation scheme of the DSL changes, it will not impact the implementation of the rules for the PTE in the backend. However, the mapping between the front-end and the backend (*i.e.,* the middle layer) will require changes if either the front-end or the backend undergoes a change. This decoupling between the components allows the framework to evolve with the evolving solution space and will be useful when FraSPA needs to be extended.

3.2 Framework Design

A set of DSLs has been developed in this research to provide a high-level interface between the end-users and FraSPA. These DSLs are meant for obtaining the specifications of desired parallel operations and ALC from the end-users. The DSLs are known as Hi-PaL (**Hi**gh-Level **Pa**rallelization **L**anguage) and DALC (**D**SL for **A**pplication-**L**evel **C**heckpointing). As it can be noticed in Figure 3-2, the process of non-invasively generating checkpointed (and hence fault-tolerant) parallel applications through FraSPA involves three steps that the end-user should undertake:

1. Identify the concurrency in the sequential application and express it using Hi-PaL.
2. Obtain the parallelized version of the application from FraSPA.
3. Analyze the parallel application and provide the specifications for CaR through the DALC.

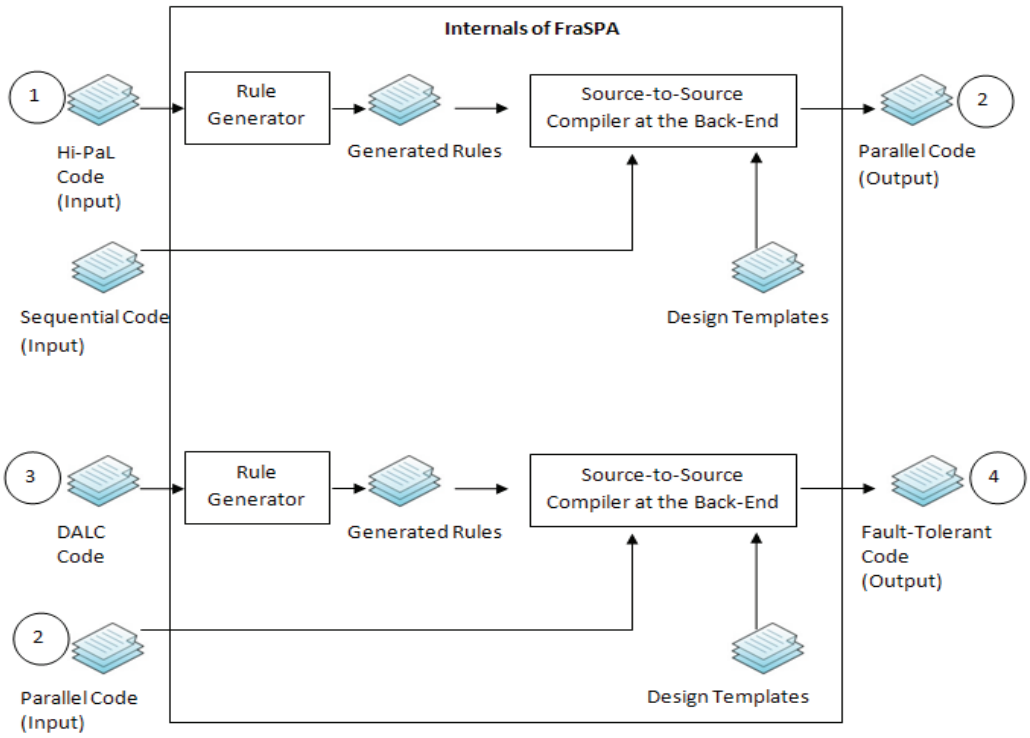

Figure 3-2- Steps for generating a checkpointed parallel application using FraSPA

To identify the concurrency in their application, the end-users should know the logic of the existing sequential application. They should know the usual terminology used in the parallel programming domain – *e.g.,* distribute, gather, and reduce. A set of guidelines has been developed for assisting the end-user in selecting the standard parallel operations available through FraSPA and expressing concurrency through Hi-PaL. In case the end-users are not interested in making their generated applications fault-tolerant via checkpointing, they might not want to proceed to step 3. However, if they wish to make the generated or existing parallel applications fault-tolerant via checkpointing, they must analyze the application and provide the CaR-specifications through DALC.

The mechanism for translating the Hi-PaL and DALC code into the actual code for parallelization and checkpointing is the same. As a summary – the rule generator translates Hi-PaL or DALC code into the rules for a SSC. The generated rules, design-templates (codified design patterns for inter-process communication, data distribution, synchronization *etc.*) and the existing (sequential or parallel) application are passed as inputs to the compiler at the back-end. The compiler modifies the AST of the existing application so that the parallel and/or checkpointed version of the application can be generated while keeping the existing application intact.

Figure 3-3 provides an overview of the internal components of FraSPA. As noted in Figure 3-3, FraSPA has a three-layered architecture comprising of front-end, middle-layer and backend. A description of each of the layers is as follows:

1. Front-End: This is the primary interface between the end-user and FraSPA. It comprises of the:

a. abstractions for expressing the specifications for explicit parallelization. (Hi-PaL)

b. abstractions for expressing the specifications for checkpointing. (DALC)

2. Middle Layer: This layer is not visible to the end-user and is used for translating the high-level abstractions obtained from the front-end into the intermediate code to be used by the backend. (Rule Generator)

3. Backend: This layer is also hidden from the end-user and is required for code instrumentation – that is, for inserting the code for parallelization and checkpointing into the existing sequential or parallel application on the basis of the intermediate code generated by the middle layer. (PTE)

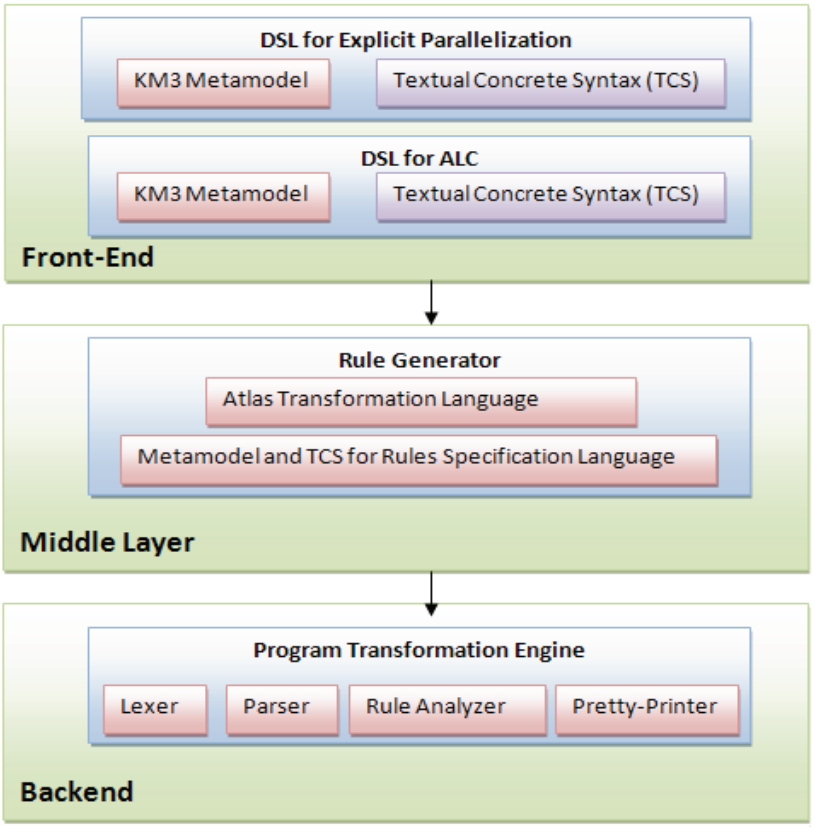

Figure 3-3- Three layered diagram of the FraSPA

In the following lines, the overall work-flow of FraSPA is explained again in context of its internal components and the layers of FraSPA. The Hi-PaL code provided by the end-user is translated into the rules by the means of a Rule Generator. These rules are analyzed by the SSC (also known as PTE) for generating and weaving the desired code for parallelization or checkpointing in the existing sequential or parallel application. The Rule Generator consists of templates written in ATL [43] and Ant Scripts. The generated rules are specific to the application that the end-user wants to parallelize and have the required C/C++/MPI code for parallelization and checkpointing.

The code in the rules is in the form of the nodes of abstract syntax tree so that the PTE, without any manual intervention, can analyze and transform the sequential application into a parallel one. The glue code, also written as Ant Scripts, is responsible for invoking the PTE and making the generated code (parallel or checkpointed) available to the end-user (step 2 and 4 in Figure 3-2). The design-templates that are a part of FraSPA are codified design patterns for inter-process communication, data distribution, and synchronization. The PTE infers which design-template to include in the process of parallelization on the basis of the generated rules and the template can be backtracked to the mappings available in the rule generator. The checkpointed parallel application thus obtained can be compiled and run like any manually-written parallel application. Figure 3-3 shows the abstractions for expressing the specifications for explicit parallelization and checkpointing that have been built in FraSPA through Hi-PaL and DALC. Instead of any particular application-domain the domains of these DSLs are explicit parallelization and ALC per se. Hi-PaL is described in Section 3.2.1, DALC is described in Section

3.2.2, and the rule generator is described in Section 3.2.3. All other components have been described in Chapter 2.

3.2.1 Hi-PaL - DSL for Parallelization

A DSL for specifying parallel computations has been developed in this research and is called Hi-PaL. Because the specifications for parallel computations can vary from application to application, different application-domains (*e.g.,* image processing, evolutionary algorithms, and stencil-based computations) were evaluated to build the key abstractions in the form of a DSL. The general structure of the Hi-PaL is shown in Figure 3-4. The mandatory structural elements of the Hi-PaL code are shown in bold-face in Figure 3-4. The italicized elements inside angular brackets are the variable structural elements of the Hi-PaL code (*e.g.,* APIs for parallelization, and statements for pattern matching). The "`&&`" operator is used for creating powerful match expressions. The Hi-PaL code will not compile if any of the mandatory keywords are missing and appropriate error messages are generated. An excerpt of the production rules of Hi-PaL grammar is shown in Figure 3-5.

```
Parallel section begins <hook type> (<hook pattern>) mapping is
<mapping type> {
<operation along with the arguments> <hook>
&& in function (<function name>)
}
```

Figure 3-4- General structure of the Hi-PaL code

It can be noticed from the grammar in Figure 3-5 that the specification for parallelization (`PARSPECS`) consists of a parallel task (`PARTASK`) and the constraints (`PARCONDITION`) for parallelization. The parallel tasks defined in this grammar consist of a subset of the standard operations provided through MPI. For example, reducing the data

(PARREDUCE, PARALLREDUCE), gathering the data from the processors (PARGATHER), and distributing the data amongst the processors (PARDISTRIBUTE). Each parallel task can be broken down further into the basic elements of the grammar. As an example, consider the rule for reducing the data. According to the Hi-PaL grammar, the specification of the reduction operation (PARREDUCE) consists of the specification of the type of reduction operation (REDTYPE), the data type of the variable being reduced (DATATYPE) and the name of the variable to be reduced (REDVARIABLE).

```
PARSPECS ::= PARTASK PARCONDITION
PARCONDITION ::= {&& HOOK PATTERN}
HOOK ::= HOOKTYPE HOOKELEMENT
HOOKTYPE ::= before|after|around|in
HOOKELEMENT ::= statement|FCT
FCT ::= function_call|function_execution
PARTASK ::= PARCOMPUTE|PARREDUCE|PARALLREDUCE|PARFOR|PARGATHER|
            PARDISTRIBUTE|PAREXCHANGE|PARBROADCAST|PARWRITE|PARREAD
PARREDUCE ::= REDTYPE DATATYPE "(" REDVARIABLE ")"
REDTYPE ::= REDUCESUM|REDUCEPRODUCT|REDUCEMINVAL|REDUCEMAXVAL
PARFOR ::= FORINITSTATEMENT; FORCOND; FOREXPRESSION
FORINITSTATEMENT ::= INITSTATEMENT|ANYSTATEMENT
INITSTATEMENT ::= FORVAR OPERATOR LIMIT
OPERATOR ::= LESSTHANEQUAL|GREATERTHANEQUAL|EQUALTO|LESSTHAN|GREATERTHAN
FORCOND ::= FORCONDPRESENT|FORNOCOND|ANYCOND
FORCONDPRESENT ::= FORVAR OPERATOR LIMIT
LIMIT ::= PARCOMPUTELIMITS
FORLOOPEXPRESSION ::= LOOPEXPRESSION|ANYEXPRESSION
LOOPEXPRESSION ::= FORVAR STRIDE
STRIDE ::= PLUSPLUS|MINUSMINUS
PARCOMPUTELIMITS ::= LOWERLIMIT|UPPERLIMIT|VARIABLEASLIMIT
```

Figure 3-5- Excerpt of the production rules in Hi-PaL

Currently, support for a subset of reduction operations is provided in FraSPA and those are MPI_SUM (`REDUCESUM`), MPI_PRODUCT (`REDUCEPRODUCT`), MPI_MAX (`REDUCEMAXVAL`), and MPI_MIN (`REDUCEMINVAL`). Additional operations can be added by extending the abstract and concrete syntax of Hi-PaL.

The end-users using Hi-PaL need not have any understanding about its grammar. The grammar-level details of Hi- Pal are only important for programmers who wish to extend the language. With Hi-PaL, without knowing anything about MPI API or its usage, end-users can specify the tasks required for parallelizing the existing sequential applications at a very high-level. Therefore, a set of Hi-PaL API has been developed for the commonly used parallel tasks like data distribution, data collection, reading or writing the data in parallel, parallelizing a for-loop, *etc.*

An excerpt of some of the Hi-PaL API and their brief description (type of MPI routine or the parallelization code associated with the API) is shown in Figure 3-6. The API-names are descriptive enough to explain their purpose. For example, **`ReduceMaxValInt`**`(<variable name>)`, means that the variable specified by `<variable name>` is of type integer and it needs to be reduced on one node (by default the node with the rank equal to zero) such that while reducing, the maximum value of the variable calculated by the individual processors is selected (MPI_MAX operation). Detailed guidelines can be provided to the end-users to help them select the appropriate API and to simplify the process of learning and using Hi-PaL. The end-users are, however, expected to be familiar with the logic of the sequential application and should be well acquainted with the concept of concurrency.

Hi-PaL API	Description
ReduceSumInt(<variable name>)	MPI_Reduce with sum operation
ReduceMaxValInt(<variable name>)	MPI_Reduce with max operation
AllReduceSumInt(<variable name>)	MPI_Allreduce with sum operation
DistributeVectorInt(<vector name>, <num of rows>**)**	MPI_Scatterv to distribute the vector
Gather2DArrayInt(<array name>, <num of rows>, <num of columns>)	MPI_Gatherv to collect the data
BroadCast2DArrayInt(<array name>, <num of rows>, <num of columns>)	MPI_Broadcast to broadcast the data
Exchange2DArrayInt(<array name>, <num of rows>, <num of columns>)	Exchange neighboring values in stencil-based computations
Parallelize_For_Loop where (<for_init_stmt>;<condition>;<stride>**)**	Parallelize for-loop with matching initialization statement, condition and stride

Figure 3-6- Excerpt of the Hi-PaL API

The syntax of Hi-PaL is similar to the syntax of other aspect languages [18]. The end-user needs to specify the hooks in the sequential application where the parallel operation needs to take effect. The complete hook definition includes the specification of hook type along with a search pattern (which is a statement in the sequential application). There are three types of hooks- *before*, *after*, and *around*- and every syntactically correct statement in a sequential application can qualify as a search pattern in Hi-PaL. In contrast to Hi-PaL, various language extensions of AOP (*e.g.,* AspectC++ and AspectC) only allow for the specification of function call, function execution, object construction, and object destruction for search purposes. The program statement specified as a hook serves as an anchor *before* or *after* which the code for parallelization needs to be woven. With the *around* hook type, the end-user gets the flexibility to delete or modify a particular statement in the sequential application. For example, if a print statement in the sequential application is not desired in the parallel version of the application, but needed as an anchor to weave some code for parallelization, the end-user can use an around type of

hook on that statement. The generated code will have the print statement replaced by the code for parallelization.

In addition to the hook, the end-user is also required to specify the desired type of data mapping in the parallel application. Data mapping means the mapping of arrays to the memories of processors and it can impact the performance of applications [70]. Some examples of the data distribution schemes are block (or linear), cyclic, block-cyclic [70]. Figure 3-7 shows a sample program written in Hi-PaL. This sample code demonstrates the method of specifying the broadcast operation on a matrix named `life` in function `main`. A one-to-one mapping of the general structure of Hi-PaL code (Figure 3-4) and a sample Hi-PaL code (Figure 3-7), is presented in Figure 3-8. This one-to-one mapping illustrates the simplicity of Hi-PaL. The standard structural elements (*e.g.,* **`Parallel section begins after`**) are going to remain the same in all the Hi-PaL programs. More examples of the usage of Hi-PaL are presented in Chapter 4.

```
Parallel section begins after ("SEED = atoi(argv[4]);") mapping is
Linear{
ParBroadCast2DArrayInt(life, M, N) after statement
("life = initMatrix<int>(life, M, N);") && in function ("main")
}
```

Figure 3-7- Sample Hi-PaL code showing the broadcast operation specification

General Structure of Hi-PaL code (Figure 3-4)	**Sample Hi-PaL code (Figure 3-7)**
Parallel section begins	**Parallel section begins**
<*hook type***>**	**After**
(<*hook pattern*>**)**	**("SEED = atoi(argv[4]);")**
mapping is	**mapping is**
<*mapping type*>	Linear
{	**{**
<*operation along with the arguments*>	**ParBroadCast2DArrayInt(**life,M,N**)**
<*hook*>	**after statement ("life = …)**
&& in function	**&& in function**
<*function name***>**	**("main")**
}	**}**

Figure 3-8- One-to-one mapping of the Hi-PaL structural elements into the sample code

3.2.2 DALC- DSL for Application-Level Checkpointing

The first step in developing any DSL is analyzing the domain (in this case ALC) for which it is being designed. During the domain analysis phase of developing the DALC, a survey of technical literature and existing implementations [49-56, 82-84] was done to obtain an overview of the terminologies and concepts related to the ALC-domain. Commonly used terms and their relationships were used to develop the domain lexicon. Commonalities and differences were observed in the process of implementing the CaR mechanism across applications in various domains and these are referred to as features from this point onward in this chapter. Some of the features in the ALC-domain and their relationships are shown as expressions in Figure 3-9.

```
ChckptgPack: all(Checkpoint, Restart)
Checkpoint: all (CheckPointCondition, CheckPointCode)
CheckPointCondition: all(Hook,Pattern,Frequency, loopVar?,CaRType)
CaRType: one-of(Centralized, Distributed, Sequential)
CheckPointCode: all(SaveVarType, saveVarArg)
SaveVarType: one-of (SaveInt, SaveDouble, SaveChar, …)
Restart: all (RestartCondition, RestartCode)
RestartCondition: all(Hook, Pattern)
RestartCode: all(ReadVarType, restartVarArg)
ReadVarType: one-of (ReadIntVarFromFile, ReadDoubleVarFromFile, …)
Hook: all (HookType, HookElement)
HookType: one-of(afterHookType, beforeHookType, aroundHookType)
HookElement: one-of(Call, Execution, Statement)
```

Figure 3-9- Excerpt of the features identified in the ALC-Domain

As shown in Figure 3-9, the feature `ChckptgPack` indicates that this DSL package allows two activities, `Checkpoint` and `Restart`. If the end-user wants to `Checkpoint` an application then the checkpoint condition, `CheckPointCondition`, and the code that should be checkpointed, `CheckPointCode`, are specified. The `CheckPointCondition`

includes the specification of the points where the code for checkpointing should be inserted (`Hook` and `Pattern`). It also includes the frequency of checkpointing (`Frequency`) and the type of CaR (`CaRType`). The expression `Hook` is made up of `HookType` and `HookElement`. Together with the `Pattern` (which is a search string), these two syntax elements identify the places in the application code where the checkpointing code should be inserted.

In case, checkpointing is required inside a loop, the name of the loop variable, `loopVar`, should also be specified. This is an optional feature and is represented by "`?`". The type of the desired CaR (`Centralized`, `Distributed` or `Sequential`) should also be specified as a part of `CheckPointCondition`. The `CheckPointCode` includes the specification of the type and name of the variable or data structure to be checkpointed. Depending upon the variable or data structure, the end-user is expected to specify a list of parameters. For example, if the end-user intends to save an integer variable, `SaveInt` is selected from the list of `SaveVarType`. The other parameters required from the end-user in this case would be the name of the variable, and the name of the file in which the variable needs to be saved.

An excerpt of the API developed for capturing the details about the variable or data-structure to be saved is presented in Figure 3-10. If the end-user intends to save a two dimensional array of type integer (specified by `SaveIntArray2D`), then apart from the name of the array and the file name, the dimension of the array also needs to be specified. Likewise, during the restart phase, as per the expression for the feature `Restart`, the end-user should specify the `RestartCondition` and the `RestartCode`. As in the case of `CheckPointCondition`, the `RestartCondition` includes the specification

of the `Hook` and `Pattern`. The `Hook` and `Pattern` are used together to identify the place where the restart code should be inserted. The `RestartCode` specification includes the description of the variable or data structure being read, the name of the variable to be initialized with the value stored in the restart file and the name of the restart file. If the restart file exists, then the variable is initialized by the value stored in the restart file, else, the program proceeds with the normal initialization process.

```
SaveInt(<variable name>, <file name>)
SaveIntArray1D(<array name>, <number of columns>, <file name>)
SaveIntArray2D(<array name>, <number of rows>, <number of columns>,
<file name>)
ReadIntVarFromFile(<variable name>, <file name>)
ReadIntArray1DFromFile(<array name>, <number of columns>, <file name>)
ReadIntArray2DFromFile(<array name>, <number of rows>, <number of
columns>, <file name>)
```

Figure 3-10- Excerpt of the API in DALC

The DALC was designed from scratch with no commonality with the existing language. However, like Hi-PaL, DALC also borrows some concepts and constructs from the AOP techniques. Similar to the concept of *advice* in AOP, the DALC has a notion of a well-defined `Hook` (shown in Figure 3-9) which is used as a handle to a specific point in the program flow. A `Hook` can be of one of the following types: *after*, *before*, and *around*. A `Hook` of type *after* has the same significance as an *after advice* in AOP. The *before* and *around* type correspond to the *before advice* and the *around advice* in AOP. It should be noted here that the *around advice* is implemented differently from its implementation in the Hi-PaL code. In Hi-PaL, the *join point* specified in the *around advice* gets entirely deleted, whereas in the DALC, the statement is preceded or succeeded with other code (see the Poisson Solver test case in Chapter 4). This advice is especially useful while

providing the code for restart mechanism because it is required to weave an if-else statement around the statement that is marked as the `Hook` for *around advice* and not to delete it. The mechanism for deleting a line might not be required for doing CaR and therefore no extra functionality for deleting a statement has been provided currently in DALC. Apart from the type, a hook definition also includes the specification of the *pointcut*.

Unlike many language extensions of AOP, in this DSL any syntactically correct program statement can be specified as a *join point*. A partial list of the type of *join points* that can be specified using this DSL are: function call, function execution, expression statement, compound statement, selection statement, and iteration statement. These different *join points* give different granularity of control to the end-user. For example, in case the *join point* is of type function execution, then the end-user gets control of the execution point of the function such that the behavior and structure of the entire code in the function body can be modified if desired. As opposed to function execution, if any one particular statement in the function needs to be modified, the *join point* should be of type statement (examples of allowed statement types are expression statement and iteration statement).

Based on the way the function execution and function call *join points* are implemented, they can differ in the scope of action. The scope of function call type of *join point* starts with the call to the function and lasts tills the program control returns from the function. The scope of function execution type of *join point* starts with the execution of the code in the body of the function and lasts till the last line of the code in the function body. The DSL keywords for expressing a *pointcut* are *call*, *execution*, and

statement along with a search pattern. An example of a `Hook` definition along with the search pattern would be:

```
around statement ("start = 0;")
```

In this example, the statement, `("start = 0;")`, serves as a *join point* of type *around*.

One of the most important steps during the design stage was choosing a structure for DALC code constructs. In DALC, the user specifies the variant features and the editor automatically generates the constant features (through the means of a wizard explained later in this chapter). As per the design, the conditions and the code for checkpointing should be provided by the end-user in the code block following the keyword `beginCheckpointing`. The conditions and the code for restart should be provided by the end-user in the code block following the keyword `beginInitialization`.

Apart from deciding the structure of the language constructs, the valid and invalid combinations of the features were also identified in the design phase. For example, any attempt to specify the code pertaining to the restart mechanism (*e.g.,* `ReadIntVarFromFile`) should not be allowed in the block following the keyword `beginCheckpointing`. Therefore, `beginCheckpointing` and `ReadIntVarFromFile` are invalid combinations of the DSL features. The valid and invalid combination of features is called configuration knowledge [19] and is required during the DSL implementation phase.

The basic structure of the DALC code for checkpointing is shown in Figure 3-11. The place-holder for the variant part, provided by the end-user, is depicted by "< >". The *`Hook`* is a statement or function call or function execution before, after or around which the checkpointing or restart functionality is desired. The *`Pattern`* of the *`Hook`* and the

`Frequency` of checkpointing, which is an integer value, are also required as a part of the CaR-specification. The "`&&`" operator is used to create a powerful expression for CaR-specifications. The **`loopVar`** shown in Figure 3-11 is an optional structural element and is used only if the variable or data structure meant to be checkpointed is inside a loop. The datastructures and variables to be checkpointed are specified within "`{`" and "`}`".

```
beginCheckpointing:
<Hook> <Pattern> && (Frequency = "<#>") &&(loopVar ="<>")
&& <CaRType>{
<checkpointing code>
}
```

Figure 3-11- Basic structure of the DALC code for checkpointing mechanism

The basic structure of the DALC code for restart is shown in Figure 3-12. The code block for restart requires the specification of *`Hook`* and *`Pattern`*. The datastructures and variables to be read from a file are specified within "`{`" and "`}`".

```
beginInitialization:
<Hook> <Pattern> {
<restart code>
}
```

Figure 3-12- Basic structure of the DALC code for restart mechanism

```
1. double computepi(int start, int end, double h) {
2. double mysum = 0.0;
3. for (int i=start; i<=end; i++) {
4. double x = h * ((double)i - 0.5);
5. mysum += 4.0 / (1.0 + x*x);
6. }
7. return h*mysum;
8. }
```

Figure 3-13- Function to compute the value of π

A simple function, `computepi`, for computing the value of pi (*i.e.,* π) using C++/MPI is shown in Figure 3-13 to illustrate the DALC code to be provided. If the variable `mysum` needs to be checkpointed after the execution of the statement at line # 5, at a frequency of every 10 iterations of the for-loop at line # 3 of Figure 3-13, then the corresponding DALC code for specifying this intention is shown in Figure 3-14. The keyword **`beginCheckpointing:`** at line # 1 of the code marks the beginning of the checkpointing block and is compulsory. The code at line # 2-4 of the Figure 3-14 expresses the checkpointing condition which in this case is to save the value of the variable `mysum`, every 10th iteration, wherever the initialization variable in the for-loop is `i`. The code at line # 6 of Figure 3-14 means that the variable named `mysum` of type `double` is being saved in a file named `restartMysum`. The iteration number is also stored in the restart file.

```
1. beginCheckpointing:
2. after statement("mysum += 4.0 / (1.0 + x*x);")
3. && (frequency = 10)
4. && (loopVar = "i" )&& (CaRType = Sequential)
5. {
6. SaveInt(i, "restartMysum")
7. SaveDouble(mysum, "restartMysum")
8. }
```

Figure 3-14- Sample DALC code for checkpointing

During the restart phase, the variable `mysum` and the starting value of iteration count, `start`, are initialized from latest checkpoint stored in the file `restartMysum`. The DALC code for specifying this intent is shown in Figure 3-15. The keyword **`beginInitialization:`** at line # 1 of the code is compulsory. As per the DSL design, if the end-user attempts to provide the CaR-specifications without providing the necessary keywords, the parser will complain about it and the code generation process will not proceed.

```
1. beginInitialization:
2. after statement ("double mysum = 0.0;")
3. {
4. ReadDoubleVarFromFile (mysum, "restartMysum")
5. ReadIntVarFromFile (start, "restartMysum")
6. }
```

Figure 3-15- Sample DALC code for restart

The DALC code needs to be translated into the source code of an existing language, usually called the base language. The DALC code in this research is transformed into the base language source code, which is C/C++, via DMS and transformation languages. First, the DALC code is translated into an intermediate code for the DMS via ATL. Using the intermediate code (generated rules), the DMS generates the code in the base language and inserts it automatically into the base application. The example code shown in Figure 3-13 is checkpointed by FraSPA, on the basis of the specifications provided in Figures 3-14 and 3-15. The output is shown in Figure 3-16. The code to save the values of the critical variables is on line # 8-13 of the code of Figure 3-16.

```
1. double computepi(int start, int end, double h) {
2.  FILE* newInputFile;
3.  /*other code*/
4.  double mysum = 0.0;
5.  for (int i=start; i<=end; i++) {
6.  double x = h * ((double)i - 0.5);
7.  mysum += 4.0 / (1.0 + x*x);
8.  if (i % 10 == 0){
9.  newInputFile = fopen("restartMysum", "w");
10.   fprintf(newInputFile, "%d", i);
11.   fprintf(newInputFile, "\n");
12.   fprintf(newInputFile, "%lf", mysum);
13.   fclose(newInputFile);
14.    }
15.    }
16.    return h*mysum;
17.   }
```

Figure 3-16- Checkpointed function to compute the value of π

With minimum effort, the DSL can be extended to add the facility to checkpoint additional data structures that are currently not covered in its present scope. In order to

promote code correctness and to reduce coding complexity, a wizard-driven GUI for DALC code generation (Figure 3-17) has been developed. The end-user can enter the CaR-specifications through the GUI instead of typing them manually. For example, the end-user can select one of the features from the list of `ReadVarType` features and provide the parameters (like variable name, restart file name). The corresponding DALC code, with the API and parameters, is generated automatically. On the basis of the selections made in the panel for providing checkpointing-specifications, the panel for restart-specifications can be generated dynamically. An outline of the workflow involved in providing the CaR-specifications is shown in the panel on the left-hand-side of the GUI. A summary page showing the CaR-specifications can be presented to the end-user in the end for the purpose of overview.

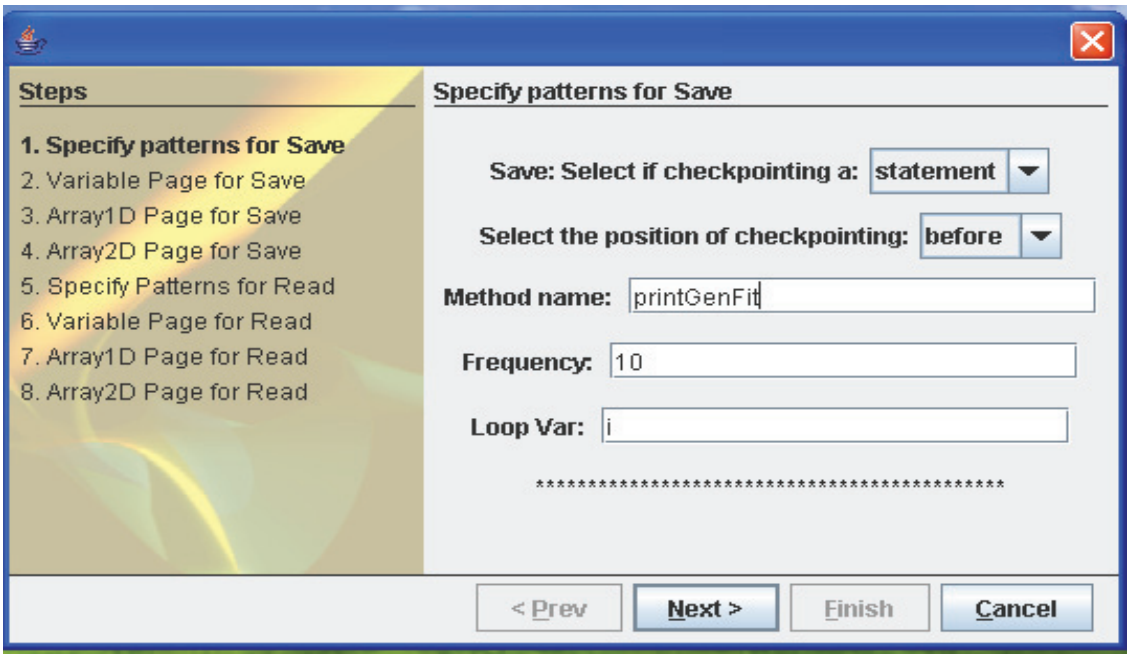

Figure 3-17- Wizard for generating the DALC code

This GUI was developed using the API and user-interface from SwingLabs, a subproject supported by Open source Java projects, an open source initiative from Sun

Microsystems and hosted at https://wizard.dev.java.net/quickstart.html. Because wizard content needs to vary dynamically (contents on the next panel depends upon the contents of the previous panel/panels), nesting of wizards within wizards was done. Input validation can be easily programmed and the process of developing this wizard-driven GUI was itself wizard-driven! This wizard can be run from any platform that has a java virtual machine installed. A similar GUI can be developed for making the process of obtaining the Hi-PaL specifications from the end-users wizard-driven. The benefits of using the DALC are summarized below:

1. Non-invasive ALC of existing applications.
2. Mitigation of the complexity associated with the usage of a PTE.
3. High-level of abstraction for source-to-source transformation.
4. Decoupling of the problem and solution space, *i.e.,* the CaR-specifications are decoupled from the actual implementation of the CaR mechanism.
5. Prevention of code tangling and thus reduction in the effort involved in software maintenance.

3.2.3 Rule Generator

A Rule Generator lies in the middle-layer of FraSPA and is required for dynamically generating the rules for the DMS in the backend. It translates the Hi-PaL or DALC code provided by the end-user into the RSL rules that the DMS can analyze. These generated rules are then used by the DMS for doing the code instrumentation – that is inserting the relevant C/C++/MPI code for parallelization and checkpointing into the existing applications. The Rule Generator not only contains the domain-knowledge [19] for generating the appropriate rules from the Hi-PaL and DALC code, but also does the task

of invoking the DMS, handling the input to DMS, and getting the output from the DMS to the end-user workspace. It comprises of ATL code, Object Constraint Language (OCL) code, metamodel and textual concrete syntax of RSL, and Ant Scripts. A description of each of these elements has been provided in Chapter 2.

3.3 Framework Implementation

Both Hi-PaL and DALC were implemented using a MDE platform called AMMA (refer Chapter 2). The AMMA platform was preferred for DSL development due to the familiarity with the same. AMMA provides KM3 and TCS for writing the abstract and concrete syntax of the DSL – while KM3 itself is like a DSL for writing new DSLs, TCS is like a grammar-template that needs to be extended. The usage of AMMA-based front-end further makes the process of extending FraSPA convenient. The extension of this MDE-based front-end might entail embedding the new grammar rules of the DSL being extended in the form of classes and templates in KM3 and TCS respectively. The already existing classes and templates will not require any modifications.

Each production rule in the Hi-PaL and DALC grammar was coded as classes in KM3 and templates in TCS. A snippet of the KM3 code for modeling the grammar rule for `PARREDUCE` (refer Figure 3-5) in Hi-PaL is shown in Figure 3-18. It can be noticed from Figure 3-18 that `ParTask` is defined as an abstract class. All the classes for specifying MPI tasks (*e.g.,* reduce, gather, and distribute) are required to extend this abstract class. The `ParReduce` class extends `ParTask` and contains references to other classes - `RedVarType`, and `RedVarArg`. Because there are multiple options available for the type of reduction operation, the class `RedVarType` is modeled as an abstract class.

Hence, the classes modeling the different types of reduction operation (*e.g.,* MPI_MAX or MPI_MIN) are required to extend the `RedVarType` class. In essence, if there are multiple values possible for a particular element in a grammar rule, then that element is modeled as an abstract class and a separate class (which can be either abstract or concrete) extending this abstract class is written for every possible value that the element can take.

```
class ParSpecs extends LocatedElement {
      reference parTask [*] container : ParTask;
      reference parCond[*] container : ParCond;
}
abstract class ParTask extends LocatedElement {
}
class ParReduce extends ParTask {
      reference redVarType container : RedVarType;
      reference varArgs[*] container : RedVarArg;
}

class RedVarArg extends LocatedElement {
      attribute argument : String;
}

abstract class RedVarType extends LocatedElement {
}
class ReduceSumInt extends RedVarType {
}
```

Figure 3-18- Excerpt of the KM3 code for modeling the `ParReduce` grammar rule

While the KM3 metamodel provides the abstract syntax of the language being developed, the concrete syntax of the language is specified in a separate model that is expressed using TCS. In a TCS model, of main interest to a language developer are the "Class templates" and the "Operator table". For every class represented in the KM3 specification, it is required to have a corresponding template definition in TCS. The "Operator table" is used for defining the syntax of DSL using operators. The terminal tokens, like separators and brackets, are a part of the TCS model. If the default lexer is not satisfactory, then the "Primitive template" in TCS can be modified as per the

requirement. If additional symbols are required then the class for "Special symbols" should be modified. An excerpt of the concrete syntax of Hi-PaL, as defined in TCS, is shown in Figure 3-19. The keyword `template` is used as a part of the definition of all the KM3 classes as templates. The name of the KM3 class (`ParReduce`) is specified along with the name of the class elements (`redVarType` and `varArgs`) defined in the KM3 model. The "," is to be used as a separator between the arguments, then the same is specified as follows:

```
{separator = ","}
```

In the template definition of `ParReduce` note the specification of `"("` and `")"`. These tokens could not be a part of KM3 model but are necessary for specifying the structure of the grammar rule and hence are a part of the TCS template definition.

```
template ParSpecs
: parTask parCond {separator = "&&"}
;

template ParTask abstract;

template ParReduce
: redVarType "(" varArgs{separator = ","} ")"
;

template RedVarArg
: argument
;

template RedVarType abstract;

template ReduceSumInt
: "ReduceSumInt"
;
```

Figure 3-19- Excerpt of the TCS code for modeling the `ParReduce` grammar rule

Apart from using AMMA platform for developing the front-end (*i.e.,* Hi-PaL and DALC), it was also used for capturing the semantics of the RSL (used in the backend by the DMS) as a metamodel. This step was required for doing the metamodel-to-metamodel translation by using the ATL (part of Rule Generator in FraSPA) in the AMMA toolsuite.

Roychoudhury *et al*. have also demonstrated the usage of AMMA platform alongside DMS and as mentioned in Chapter 2, the design of FraSPA is influenced by the design of their framework for generic aspect weaving [37, 81]. In summary, the fundamental nature of Hi-PaL and DALC used in the front-end, and the RSL used in the backend is captured through metamodels written in KM3 and TCS. The metamodels for the Hi-PaL and DALC in the front-end are known as "source metamodels" whereas the metamodel for the RSL to be used in the backend is called the "target metamodel". The high-level specifications provided by the end-user, in the form of Hi-PaL or DALC code, are first injected into the DSL metamodel. These high-level specifications can be considered as a terminal model in the MDE parlance. The specifications are validated against the metamodel during the process of injection. With the help of the ATL transformations and Ant Scripts, the code injected in the DSL metamodel is translated into the RSL terminal model (RSL rules). The process of obtaining the RSL terminal model from the RSL metamodel is called extraction. The RSL rules thus generated are used by the DMS for weaving the parallelization or checkpointing code into the existing application. All these steps result in the transformed code (parallel or fault-tolerant) and the complete workflow of the process of transformation through models is pictorially shown in Figure 3-20.

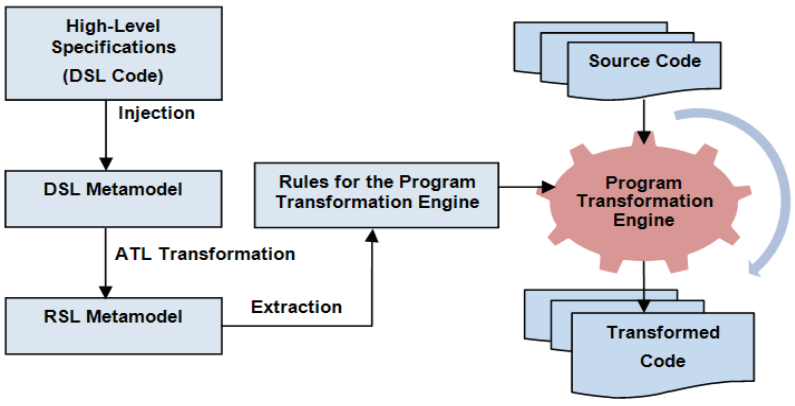

Figure 3-20- Extraction and injection of models in FraSPA

The mapping of the elements of the DALC into the metamodel elements is pictorially shown in Figure 3-21. The KM3 classes corresponding to the syntactic elements of the DALC code are shown on the left hand-side of Figure 3-21. There are three concrete classes for the hook type (**before**, **after**, **around**) and the DALC code for checkpointing shown in Figure 3-21 uses an **around** type of hook on a **statement**. Each variable or data structure that should be checkpointed is mapped into a **ChkStmt** class. The data members of this class are the type and the name of the variable or data structure to be used, and additional parameters like the name of the restart file or the dimensions of the array. In the example code in Figure 3-21, the line `SaveInt(k, restart)` of the DALC code implies that a variable of type integer is being checkpointed and the variable type is mapped into the KM3 class called as **SaveInt**. The parameter `k` is the name of the variable that should be checkpointed and the parameter **restartK** is the name of the file in which the checkpointing data should be saved.

The set of ATL rules (ATL + OCL code), Ant Scripts, and RSL metamodel make the Rule Generator. A snippet of the ATL rule is shown in Figure 3-22. As can be noticed from Figure 3-22, the ATL rule consists of the description of the source metamodel (DSL) and the target metamodel (RSL) along with the mapping of the syntactic elements from the source metamodel to the target metamodel. The source and target metamodels are specified as **from** and **to** in the rule. Each type of RSL rule (*e.g.,* for modifying a for-loop, for setting up the MPI environment, and for performing the gather operation), is modeled as a separate ATL rule using the elements of ATL syntax and OCL expressions.

Figure 3-21- DSL code mapped into KM3 model

The ATL rule snippet that is shown in Figure 3-22 is meant for generating the RSL rule for automating the insertion of the MPI code for reduce operation in the existing sequential application. The compulsory elements of the RSL rule (*e.g.,* '`statement_seq`' and '`add_var`') are hard-coded in the ATL rule. The variable parts that are application-specific are automatically derived from the terminal model (or the Hi-PaL code) provided by the end-user. It can be observed from the code snippet shown in Figure 3-22 that the OCL expressions are used for traversing the nodes of the terminal model for obtaining the values of the variables in the ATL rule. The OCL expressions shown in Figure 3-22 are meant to derive the name of the variable to be reduced from the terminal model. A suffix "`_Fraspa`" is added to the name thus obtained from the front-end specifications. The data type of the variable to be reduced is also derived from the

terminal model. Both these derived values are used for declaring a variable in which the global value of the MPI operation (*e.g.,* MPI_SUM or MPI_MIN) is stored during the reduce operation.

```
module PSDL2RSL;
create OUT : RSL from IN : PDSL;
rule PSDL2RSL {
   from
    s : PDSL!PDSL
   to
    t : RSL!RSL (
        domain <- dom,
        rslelems <- Sequence {pat1, expat1, rule1, pat2, pat3, expat2, rule2},
        ruleset <- rs
    ),
    dom : RSL!Domain(
        dname <- 'Cpp'
    ),
    rs : RSL!RuleSet (
        rsname <- 'r',
        rname <- Sequence {'extend_decl', 'add_statements'}
    ),
    pat1 : RSL!Pattern(
        phead <- ph,
        ptoken <- 'statement_seq',
        ptext <- pt
    ),
    ph : RSL!PatternHead (
        name <- 'add_var'
    ),
    pt : RSL!SimplePatternText (
        ptext <- s.parSpecs->iterate(parSpec; c : String = ''| c +
      if
       (parSpec.parTask->first().oclIsKindOf(PDSL!ParReduce))
     then
       if
        (parSpec.parTask->first().redVarType.oclIsTypeOf(PDSL!ReduceMaxValInt)
        or parSpec.parTask->first().redVarType.oclIsTypeOf(PDSL!ReduceSumInt)
        or ...)
       then
        '\\>Cpp\\:[simple_declaration = decl_specifier_seq
             init_declarator_list\';\'] int
             \\>Cpp\\:[declarator_id = id_expression]'
               + parSpec.parTask->first().varArgs->first().argument + '_Fraspa
             \\<\\:declarator_id ;
        \\<\\:simple_declaration'
       else if (...
```

Figure 3-22- ATL code snippet

The Ant Scripts are used in FraSPA for saving the KM3 model in the Ecore format, transforming the source model to target model on the basis of the specified ATL rule, serializing the target model into text, and for debugging purposes. They are also

used as the glue code for copying the files (RSL rules and sequential code) from AMMA platform to the required folders in the DMS installation, for invoking the DMS engine for code weaving, and for copying the generated parallel code back to AMMA platform. Therefore, the Ant Scripts are used for automating the complete workflow in FraSPA. A snippet of the RSL rule generated by FraSPA for extending the variable declaration section in the existing sequential application is shown in Figure 3-23. An example of using this new variable can be in storing the global result of performing a reduce operation.

```
default base domain Cpp~VisualCpp6.
pattern add_var() : statement_seq =
"\>Cpp~VisualCpp6\:[simple_declaration = decl_specifier_seq
   init_declarator_list';'] int
    \>Cpp~VisualCpp6\:[declarator_id = id_expression]
          norm_Fraspa
    \<\:declarator_id
  ;
 \<\:simple_declaration ".

external pattern addVars(tu : translation_unit, stmt_seq : statement_seq) :
translation_unit
= 'addVars' in domain Cpp~VisualCpp6.

rule extend_decl(tu : translation_unit):
translation_unit->
translation_unit
=
tu ->
addVars(tu, add_var())
if tu ~= addVars(tu, add_var()).
```

Figure 3-23- RSL rule snippet

3.4 Summary

The process of developing Hi-PaL and DALC was explained in this chapter. The sample code for KM3 metamodel and TCS grammar for Hi-PaL are presented in Appendix A and that for DALC are presented in Appendix B. Sample ATL rule is shown in Appendix C. A sample of RSL rule generated by FraSPA is presented in Appendix D and samples of PARLANSE code are provided in Appendix E. The DSLs required in this

research were written using AMMA platform and are used as the front-end of FraSPA. The middle-layer of FraSPA comprises of the Rule Generator (ATL rule templates, Ant Scripts and the RSL metamodel). For both the DSLs developed in this research, the target RSL metamodel remained the same and is similar to the one developed by Roychoudhury *et al.* [37, 81]. The DMS was used for weaving the code for parallelization and checkpointing in the existing applications. The current implementation of the FraSPA supports the transformation of code written in C/C++ but it can be extended for code written in other languages as well. Support for a limited set of C/C++ grammar rules and MPI API is provided in the current implementation of FraSPA.

Besides DMS, there are other PTEs that are available today. Some of them are ROSE [35], TXL [85], Stratego [86], and ASF + DSF [87]. DMS was preferred over these open-source projects because it is a mature and scalable tool that has tool support (Lexer, Parser, Pretty Printer, Rule Analyzer) available for over 20 domains. Due to the decoupling between the components in the different layers of the FraSPA, the DMS can be swapped with a better PTE should there be one available in future. As mentioned in Section 3.1, changing the PTE in the backend would not necessitate any changes in the front-end. However, it will require that the new mappings are written between the front-end and the backend, thereby, necessitating changes in the rule generator component.

CHAPTER 4

EXPERIMENTAL EVALUATION

The applications generated through FraSPA were evaluated for performance, accuracy, scalability, and fault-tolerance (through checkpointing). The framework itself was evaluated for the amount of reusable code components and the amount of effort involved in generating applications belonging to various domains. The aim of the experiments run in this research was to compare the code generated through FraSPA with its manually-written counterpart. The test cases used for evaluating FraSPA are described in Section 4.1 of this chapter. The experimental set-up and evaluation of FraSPA are described in Section 4.2. The results and analysis are presented in Section 4.3. A general discussion and summary are presented in Section 4.4.

4.1 Test Cases

The various test cases used to study the behavior of FraSPA are presented in the following subsections. These test cases had already existing manual implementations of sequential, parallel, and checkpointed versions written in C/C++/MPI. Only those test cases were selected that added value in highlighting the usability of FraSPA and the features that are currently available (*e.g.,* gathering, distributing, and reducing the data) – that is, the selected test cases depict diverse combination of parallel operations. The existing sequential applications were used to embed the code for parallelization and

checkpointing such that the generated parallel versions and checkpointed versions had communication patterns similar to their manually-written counterparts. The selected test cases were useful in evaluating the performance, accuracy, scalability, and reliability of the applications generated through FrasPA, and successfully demonstrated these following properties:

- it is application-domain-neutral,
- it reduces the programmer effort through code reuse, and
- it reduces the application development time by reducing the number of lines of code that the programmer has to write.

For parallelizing the applications through FraSPA, the programmers should be familiar with the logic of the corresponding applications, must be aware of the hotspots for parallelization (they can profile their applications for this purpose), and must express the specifications for the desired parallelization through the Hi-PaL code. Likewise, for automatically checkpointing the applications, the programmers must identify the main data structures or variables from which the entire execution state of the application can be recreated in case of a failure. Since the checkpointing approach developed in this research falls under the category of ALC, it involves saving the state of the critical data structures or variables to a secondary storage medium, and it can incur extra run-time overheads. Therefore, the frequency at which the checkpoint is taken is also important and should be specified by the programmer. The place in the application where the checkpoint should be taken can affect the accuracy of the results in case of the restart. Therefore, for doing both parallelization and checkpointing (for fault-tolerance) through FraSPA, the programmer must specify where code insertions are performed. The process of making

the applications fault-tolerant via checkpointing is demonstrated in three test cases. Only three test cases are needed because there are no significant differences in the process of checkpointing the other test cases. A summary of parallel operations applied on each test case and whether or not the test case was made fault-tolerant through checkpointing is presented in Table 4-1. The communication patterns exhibited by the selected test cases are also shown in Table 4-1. The classification of test cases as per the communication pattern they exhibit is done according to the guidelines provided in [88]. A brief description of these patterns is as follows:

- Embarrassingly Parallel: This pattern describes the concurrent execution of a collection of independent tasks (having no data dependencies). Implementation techniques include parallel loops and Manager-Worker.
    - Parallel Loop: If the computation fits the simplest form of the pattern such that all tasks are of the same size, and are known a priori then they can be computed by using a parallel loop that divides them as equally as possible amongst the available processors.
    - Manager-Worker: Also known as task queue, this pattern involves two set of processors – Manager and Worker. There is only one Manager that creates and manages a collection of tasks (task queue) by distributing it amongst the available Workers and collecting the results back from them.

- Mesh: Also known as “stencil-based computations”, this pattern involves a grid of points in which new values are computed for each point in the grid on the basis of the data from the neighboring points in the grid.
- Pipeline: This pattern involves the decomposition of the problem into ordered group of data-dependent tasks. The ordering of tasks does not change during the computation.
- Replicable: This pattern involves multiple sets of operations that need to be performed using a global data structure and hence having dependency. The global data is replicated for each set of operations and after the completion of operations, the results are reduced

Table 4-1- Parallel operations applied on the test cases

Test Case	**Parallel Operation**	**Communication Pattern**	**Checkpointing**
Prime Number	For-Loop, Reduce Sum, Reduce Max	Embarrassingly Parallel (Parallel Loop)	No
Circuit Satisfiability	For-Loop, Reduce Sum	Embarrassingly Parallel (Parallel Loop)	Yes
Poisson Solver	Exchange, AllReduce	Mesh	Yes
Game of Life	Distribute, Exchange, AllReduce	Mesh	No
Image Processing	Distribute, Reduce Sum, Gather	Manager-Worker	No
Mandelbrot Set	Distribute, Gather	Manager-Worker	No
Genetic Algorithm	Distribute, Gather, Reduce Sum, For-Loop	Pipeline, Replicable	Yes

4.1.1 Prime Number Generation

The Sieve of Eratosthenes algorithm is used for finding the prime numbers between 1 to N, where N is any natural number. The code snippet in Figure 4-1 is from

the sequential implementation of the application. If the intention is to parallelize the for-loop at line # 4 of Figure 4-1, find the global sum of the variable `pc`, and find the largest prime number in a given range, then the required Hi-PaL code is shown in Figure 4-2. As can be noticed from the Hi-PaL code, the for-loop with the `(n=1; n<=LIMIT; n++)` pattern should be parallelized and two reduce operations are required (lines #3-4 of Figure 4-2), one to find the global sum of the variable `pc` and the other one to find the largest prime number in a given range. The line # 1 of the code in Figure 4-2 means that the code for setting up the MPI environment should be inserted after the statement `t1=gettime();` in the sequential code (line # 2 in Figure 4-1). The code snippet from the generated parallel code is shown in Figure 4-3. The code for all the required variables, API, and files to include (*e.g.,* mpi.h) is generated automatically from the Hi-PaL code in Figure 4-2. To avoid any naming conflicts between the generated and user-defined variables, the generated variables have a different namespace (***_Fraspa**). The parallelized for-loop is shown at line # 6 of Figure 4-3 and the statements for computing the values of **lower_limit_Fraspa** and **upper_limit_Fraspa** are inserted automatically but not shown in the code snippet presented here.

```
1.    //other code
2.    t1= gettime();
3.    pc=0;
4.    for (n=1; n<=LIMIT; n++) {
5.      if (isprime(n)) {
6.       pc++;
7.       foundone = n;
8.       printf("%d\n",foundone);
9.      }
10.     if (n>2){
11.      n=n+1;
12.     }
13.   }
14.   t2= gettime();
15.   //other code
```

Figure 4-1- Code snippet of the sequential prime number generation application

```
1. Parallel section begins before ("t1=gettime();") mapping is
   Linear {
2. Parallelize_For_Loop where (n=1; n<=LIMIT; n++)
   after statement ("pc=0;") && in function ("main");
3. ReduceSumInt(pc) in function ("main");
4. ReduceMaxValInt(foundone) in function ("main")
5. }
```

Figure 4-2- Hi-PaL code for parallelizing the prime number generation application

```
1. //other code. Files included & Variable declaration section
   //extended.
2. t1= MPI_Wtime();
3. pc=0;
4. MPI_Init(NULL, NULL);
5. //other code
6.    for (n = lower_limit_Fraspa; n <= upper_limit_Fraspa;n++)
7.    {
8.      if (isprime(n)) {
9.          pc++;
10.         foundone = n;
11.         printf("%d\n",foundone);
12.     }
16.     if (n>2){
17.      n=n+1;
18.     }
13.   }
14.   {
15.    MPI_Reduce(&pc, &pc_Fraspa, 1, MPI_INT, MPI_SUM,…);
16.    MPI_Reduce(&foundone, &foundone_Fraspa, 1, MPI_INT,
                  MPI_MAX,…);
17.
18.    pc = pc_Fraspa  ;
19.    foundone = foundone_Fraspa;
20.   }
21.   t2= MPI_Wtime();
22.   //other code
```

Figure 4-3- Code snippet of the generated parallel prime number generation application

4.1.2 Circuit Satisfiability

This embarrassingly parallel application is adapted from Michael Quinn's book on "Parallel programming in C with MPI and OpenMP" [89]. The application simulates the actual circuit and determines whether a combination of inputs to the circuit of logical gates produces an output of 1. The application involves an exhaustive search of all the possible combinations of the specified number of bits in the input. For example, for a

circuit having 30 bits of input, the search space would involve 2^{30} combinations of the bits, which is 1,073,741,824 possibilities. A code snippet from the sequential version of the application is shown in Figure 4-4. In this code snippet, the computation being done in the for-loop (line # 6-13) can be done in parallel and the results of the computation can be reduced after the for-loop. If the programmer wants to begin the parallel section after the statement on line # 3 of Figure 4-4, wishes to parallelize the for-loop on line # 6, and wants to reduce the results before line # 14, then this intention is expressed through the Hi-PaL code shown in Figure 4-5. As noted in Figure 4-5, the `&&` operator is used to create a powerful match expression in line # 2. If the programmer does not specify the function name and place in the code where the for-loop needs to be parallelized, then everywhere (in any function or module) a matching for-loop is found in the application, it will get parallelized. This scenario will happen in the case of statements that cut through multiple modules. A snippet from the generated parallel code is shown in Figure 4-6. Like the previous test case, the code for all the required variables, API, and files to include (*e.g.,* mpi.h and design templates) is generated automatically from the Hi-PaL specifications.

```
1. //other code
2. ilo = 0;
3. ihi = pow(2, n);
4. solution_num = 0;
5. t1 = gettime();
6. for ( i=ilo; i<ihi; i++ ){
7.   //other code
8.   value = circuit_value ( n, bvec );
9.   if ( value == 1 ) {
10.     solution_num = solution_num + 1;
11.     //other code
12.   }
13.   }
14. t2 = gettime();
```

Figure 4-4- Code snippet from the sequential circuit satisfiability application

```
1. Parallel section begins after ("ihi=pow(2,n);") mapping is
   Linear {
2.   Parallelize_For_Loop where (i=ilo; i<ihi; i++)
3.   after statement ("ihi=pow(2,n);") && in function "main");
4.   ReduceSumInt(solution_num) before statement
5.   ("t2=gettime();") && in function ("main")
6. }
```

Figure 4-5- Hi-PaL code for parallelizing the circuit satisfiability application

```
1. //other code. Files included & Variable declaration section
   //extended.
2.    ilo = 0;
3.    ihi = pow(2, n);
4.    MPI_Init(NULL, NULL);
5.    MPI_Comm_size(MPI_COMM_WORLD, &size_Fraspa);
6.    MPI_Comm_rank(MPI_COMM_WORLD, &rank_Fraspa);
7.    lower_limit_Fraspa = rank_Fraspa *((ihi - ilo)…;
8.    upper_limit_Fraspa=((rank_Fraspa==(size_Fraspa - 1))?...;
9.    solution_num = 0;
10.   t1 = MPI_Wtime();
11.   for (i=lower_limit_Fraspa; i<=upper_limit_Fraspa;i++){
12.         value = circuit_value ( n, bvec );
13.         if ( value == 1 ) {
14.            solution_num = solution_num + 1;
15.            //other code
16.         }
17.   }
18.   MPI_Reduce(&solution_num,&solution_num_Fraspa,...)
19.   solution_num = solution_num_Fraspa;
20.   t2 = MPI_Wtime();
```

Figure 4-6- Code snippet from the generated parallel circuit satisfiability application

The parallel code in Figure 4-6 can be made fault-tolerant by inserting the CaR mechanism in it. The first step towards achieving this goal is to identify the critical variables in the application from which the complete execution state can be recreated. The critical variables for this application are **`upper_limit_Fraspa`**, the iteration number which is **`i`**, and the number of solutions found (which is `solution_num` in the code). For brevity, `solution_num` is not considered for the illustration of the checkpointing technique and only the values of **`upper_limit_Fraspa`** and **`i`** are being shown to be saved. It is best to insert the checkpointing code after lines # 8 and 14 of the code in Figure 4-6. The DSL code for checkpointing this application is shown in Figure 4-7.

Because this involves a **Distributed** checkpoint, each processor is responsible for saving the state of the critical variables in separate files. The restart code in Figure 4-8 illustrates the usage of **after statement** type of Hook. The instrumented code is shown in Figure 4-9. As can be noticed from this code, the file names for saving and reading the critical variables are generated dynamically for the **Distributed** CaR type by calling the function named **fileName_Fraspa**.

```
1. beginCheckpointing:
2. after statement("upper_limit_Fraspa =((rank==(size - 1))?…")
3. && (frequency = 1) && (CaRType = Distributed){
4.   SaveLong (upper_limit_Fraspa, restartUpperLimit)
5. }

6. beginCheckpointing:
7. after statement("solution_num = solution_num + 1;") &&
   (frequency = 100) && (loopVar="i") && (CaRType = Distributed){
8.   SaveLong (i, restartLowerLimit)
9. }
```

Figure 4-7- Checkpointing specifications for circuit satisfiability application

```
1. beginInitialization:
1. before statement ("lower_limit_Fraspa = rank*((ihi - ilo)… "){
2.    ReadLongVarFromFile(ilo, "restartLowerLimit")
3.    ReadLongVarFromFile(ihi, "restartUpperLimit")
4. }
```

Figure 4-8- Restart specifications for circuit satisfiability application

4.1.3 Poisson Solver

Solving second-order partial differential equations is one of the most common computational tasks performed in the Computational Fluid Dynamics (CFD) domain. The Poisson Solver is a representative application that illustrates the communication and computation patterns in a typical CFD application. In this particular case-study we are considering a solution to a two-dimensional Poisson problem with a five-point stencil

[90, 91]. The solution involves iterative computation of values at each point in the computational domain using the neighboring cells from the previous iteration. This is done till the convergence criterion is satisfied.

```
1.    //other code. Files included & Variable declaration section
      //extended.
2.    char fname1[20] = "restartLowerLimit";
3.    char fname2[20] = "restartUpperLimit";
4.    char *addString1, *addString2;
5.    //other code
6.    ilo = 0;
7.    ihi = pow(2, n);
8.    MPI_Init(NULL, NULL);
9.    MPI_Comm_size(MPI_COMM_WORLD, &size_Fraspa);
10.   MPI_Comm_rank(MPI_COMM_WORLD, &rank_Fraspa);
11.   addString1 = fileName_Fraspa(fname1, rank_Fraspa);
12.   addString2 = fileName_Fraspa(fname2, rank_Fraspa);
13.   inputfile1 = fopen(addString1, "r");
14.   if(inputfile1 !=NULL){
15.      fscanf(inputfile1 , "%lld", &ilo);
16.      fclose(inputfile1);
17.   }
18.   inputfile2 = fopen(addString2, "r");
19.   if(inputfile2 !=NULL){
20.      fscanf(inputfile2, "%lld", &ihi);
21.      fclose(inputfile2);
22.   }
23.   lower_limit_Fraspa = rank_Fraspa *((ihi - ilo)…;
24.   upper_limit_Fraspa=((rank_Fraspa==(size_Fraspa - 1))?...;
25.   solution_num = 0;
26.   t1 = MPI_Wtime();
27.   for (i=lower_limit_Fraspa; i<=upper_limit_Fraspa;i++){
28.      value = circuit_value ( n, bvec );
29.      if ( value == 1 ) {
30.        solution_num = solution_num + 1;
31.        //other code
32.      }
33.   }
34.   MPI_Reduce(&solution_num,&solution_num_Fraspa,...)
35.   solution_num = solution_num_Fraspa;
36.   t2 = MPI_Wtime();
```

Figure 4-9- Code snippet of the checkpointed circuit satisfiability application

A code snippet of the sequential version of Poisson Solver is shown in Figure 4-10. For parallelizing this application, the matrices `a` and `b` should be blocked and the cells at the border of the blocks should exchange values with their neighbors after the initialization is complete. The neighboring blocks should exchange the value of the

border-cells of matrix `b` in every iteration of the for-loop starting at line # 10 of Figure 4-10. Apart from exchanging values, the value of the `norm` computed in every iteration of the for-loop should also be reduced. All these steps for parallelization are specified through the Hi-PaL code in Figure 4-11. A code snippet from the generated code is shown in Figure 4-12. The code inserted by FraSPA is shown in bold-face. The code for calling including the function template for exchanging the desired values of the matrices is at line # 21, 22 and 27 of Figure 4-12.

```
1. //other code
2. NTIMES = atoi(argv[3]);
3. a = allocMatrix<double>(a, M, N);
4. b = allocMatrix<double>(b, M, N);
5. f = allocMatrix<double>(f, M, N);
6. start = 0;
7. //other code
8. printMatrix<double>(a, M, N);
9. t1 = gettime();
10.   for (k = start; k < NTIMES && norm >= tolerance; k++) {
11.     b = compute(a, f, b, M, N);
12.     ptr = a;
13.     a = b;
14.     b = ptr;
15.     norm = normdiff(b, a, M, N);
16.   }
17. t2 = gettime();//other code
```

Figure 4-10- Code snippet from the sequential version of the Poisson Solver

```
1. Parallel section begins after ("NTIMES = atoi(argv[3]);")
   mapping is Linear{
2. ParExchange2DArrayDouble (a, M, N) before statement
      ("printMatrix<double>(a, M, N);") && in function ("main");
3. ParExchange2DArrayDouble (b, M, N) before statement
      ("printMatrix<double>(a, M, N);") && in function ("main");
4. ParExchange2DArrayDouble (b, M, N) after statement
      ("b=compute(a, f, b, M, N);") && in function ("main");
5. AllReduceSumInt(norm) after statement
      ("norm = normdiff(b, a, M, N);") && in function ("main")
6. }
```

Figure 4-11- Hi-PaL code snippet for parallelizing the Poisson Solver

To make the code shown in Figure 4-12 fault-tolerant, the programmer needs to provide the CaR specifications via DALC as shown in Figure 4-13. The critical variables

and data structures for this application are matrices `a` and `f`, the number of iterations which is `k`, and the `norm`. The checkpointing code in this application should be inserted before line # 26 of the code in Figure 4-12. The code at line # 2 of Figure 4-13 is the `Hook` specification (line # 26 of the code in Figure 4-12) and is required for pattern matching in the abstract syntax tree of the application code. The intent of writing the variables and data structures to appropriate files is expressed in line # 5-8 of Figure 4-13.

```
1. //other code
2. NTIMES = atoi(argv[3]);
3. MPI_Init(NULL, NULL);
4. MPI_Comm_size(MPI_COMM_WORLD, &size_Fraspa);
5. MPI_Comm_rank(MPI_COMM_WORLD, &rank_Fraspa);
6. create_2dgrid(MPI_COMM_WORLD, &comm2d_Fraspa,…);
7. create_diagcomm(MPI_COMM_WORLD, size_Fraspa, …);
8. rowmap_Fraspa.init(M, P_Fraspa, p_Fraspa);
9. colmap_Fraspa.init(N, Q_Fraspa, q_Fraspa);
10.   myrows_Fraspa = rowmap_Fraspa.getMyCount();
11.   mycols_Fraspa = colmap_Fraspa.getMyCount();
12.   M_Fraspa = M;
13.   N_Fraspa = N;
14.   M = myrows_Fraspa;
15.   N = mycols_Fraspa;
16.   a = allocMatrix<double>(a, M, N);
17.   b = allocMatrix<double>(b, M, N);
18.   f = allocMatrix<double>(f, M, N);
19.   start = 0;
20.   //other code
21.   a = exchange<double>(a, myrows_Fraspa + 2, mycols_Fraspa +
      2,…);
22.   b = exchange<double>(b, myrows_Fraspa + 2, mycols_Fraspa +
      2,…);
23.   printMatrix<double>(a, M, N);
24.   t1 = MPI_Wtime();
25.   for (k = start; k < NTIMES && norm >= tolerance; k++) {
26.    b = compute(a, f, b, M, N);
27.    b = exchange<double>(b, myrows_Fraspa + 2, mycols_Fraspa
       2,…);
28.    ptr = a;
29.    a = b;
30.    b = ptr;
31.    norm = normdiff(b, a, M, N);
32.    MPI_Allreduce(&norm, &norm_Fraspa, 1, MPI_INT, MPI_SUM,…);
33.    norm = norm_Fraspa;
34.   }
35.   t2 = MPI_Wtime();
36.//other code
```

Figure 4-12- Code snippet from the generated parallel version of the Poisson Solver

```
1. beginCheckpointing:
2. before statement ("b = compute(a, f, b, M, N);")
3. && (frequency = 10)
4. && (loopVar="k") && (CaRType = Centralized){
5.   SaveDoubleArray2D(a,M,N,restartA)
6.   SaveDoubleArray2D(f,M,N,restartF)
7.   SaveDouble (norm,restartNorm)
8.   SaveInt (k,restartK)
9. }
```

Figure 4-13- DALC code snippet for describing checkpointing in Poisson Solver

The DALC code for specifying the restart mechanism for this application is shown in Figure 4-14. Lines # 3-4 of the code imply that the matrices `a` and `f` should be initialized from the values read from the files `restartA` and `restartF`. In case these restart files are not present, the matrices are initialized by calling `initMatrix <double> (a, N, N, value)` and `initMatrix<double>(f, N, N, value)` respectively. The inserted CaR code is shown in lines # 19-28 and 31-50 of Figure 4-15.

```
1. beginInitialization:
2. around statement ("start = 0;"){
3.   ReadDoubleArray2DFromFile (a,M,N,"restartA") |
          initMatrix <double>(a, M, N, value)
4.   ReadDoubleArray2DFromFile (f,M,N,"restartF") |
          initMatrix<double>(f, M, N, value)
5.   ReadDoubleVarFromFile (norm,"restartNorm")
6.   ReadIntVarFromFile (start,"restartK")
7. }
```

Figure 4-14- DALC code snippet for describing the restart mechanism in Poisson Solver

4.1.4 Game of Life

The Game of Life is a board game that consists of a two-dimensional array of cells. Each cell can hold an organism and has eight neighboring cells (left, right, top, bottom, top-left, bottom-right, top-right, and bottom-left). Each cell can be in two states: alive or dead. The game starts with an initial state and cells either live, die or multiply in the next iteration (generation) according to the following rules:

1. If a cell is alive in the current generation, then depending on the state of its neighbors, in the next generation the cell will either live or die based on the following conditions:
    - Each cell with one or no neighbor dies, as if by loneliness.
    - Each cell with four or more neighbors dies, as if by overpopulation.
    - Each cell with two or three neighbors survives.
2. If a cell is dead in the current generation but if there are exactly three neighbors alive then it will change to the alive state in the next generation, as if the neighboring cells gave birth to a new organism.

The rules of the game are applied at each iteration (generation) so that the cells evolve or change state from generation to generation. Also all cells are affected simultaneously in a generation (*i.e.,* for each cell you need to use the value of the neighbors in the current iteration to compute the values for the next generation). This application is an example of stencil-based computation. A code snippet from the sequential version of the application is shown in Figure 4-16. One way to parallelize this application would be to block and distribute the two-dimensional life matrix amongst the available processors and let each of the processors do the computation on their respective block of the matrix. After distributing the life matrix, it is required to exchange the initial values of the cells at the borders of the neighboring blocks in every iteration, and collect the information about the number of cells that are alive in each block of the matrix.

```
1. //other code
2. NTIMES = atoi(argv[3]);
3. MPI_Init(NULL, NULL);
4. MPI_Comm_size(MPI_COMM_WORLD, &size_Fraspa);
5. MPI_Comm_rank(MPI_COMM_WORLD, &rank_Fraspa);
6. create_2dgrid(MPI_COMM_WORLD, &comm2d_Fraspa,…);
7. create_diagcomm(MPI_COMM_WORLD, size_Fraspa, …);
8. rowmap_Fraspa.init(M, P_Fraspa, p_Fraspa);
9. colmap_Fraspa.init(N, Q_Fraspa, q_Fraspa);
10.myrows_Fraspa = rowmap_Fraspa.getMyCount();
11.mycols_Fraspa = colmap_Fraspa.getMyCount();
12.M_Fraspa = M;
13.N_Fraspa = N;
14.M = myrows_Fraspa;
15.N = mycols_Fraspa;
16.a = allocMatrix<double>(a, M, N);
17.b = allocMatrix<double>(b, M, N);
18.f = allocMatrix<double>(f, M, N);
19.if (!restart) {
20.   start = 0;
21.   initMatrix<double>(a, M, N, value);
22.   initMatrix<double>(f, N, N, value);
23.} else { // read a, f, norm and start from restart file
24.   readMatrix(a,M,N, "restartA");
25.   readMatrix(f,N,N, "restartF",);
26.   readVar(&norm, "restartNorm");
27.   readVar(&start, "restartK");
28.}
29./*other code*/
30.for (k = start; k < NTIMES && norm >= tolerance; k++){
31. if(k % 10 == 0){
32.   inputfile1 = fopen("restartA", "w");
33.   inputfile2 = fopen("restartF", "w");
34.   inputfile3 = fopen("restartNorm", "w");
35.   inputfile4 = fopen("restartK", "w");
36.   for (ii = 0; ii < M; ii++){
37.     for (jj = 0; jj < N; jj++){
38.       fprintf(inputfile1, "%lf  ", a[ii][jj]);
39.       fprintf(inputfile2, "%lf  ", f[ii][jj]);
40.     }
41.     fprintf(inputfile1, "\n");
42.     fprintf(inputfile2, "\n");
43.   }
44.   fprintf(inputfile3, "%lf  ", norm);
45.   fprintf(inputfile4, "%d  ", k);
46.   fclose(inputfile1);
47.   fclose(inputfile2);
48.   fclose(inputfile3);
49.   fclose(inputfile4);
50. }
51. b = compute(a, f, b, M, N);
52. b = exchange<double>(b, myrows_Fraspa + 2, mycols_Fraspa + 2,…);
53. ptr = a;
54. a = b;
55. b = ptr;
56. norm = normdiff(b, a, M, N);
57. MPI_Allreduce(&norm, &norm_Fraspa, 1, MPI_INT, MPI_SUM,…);
58. norm = norm_Fraspa;
59.}
60.t2 = MPI_Wtime();
61.//other code
```

Figure 4-15- Code snippet of the checkpointed Poisson Solver

```
1. //other code
2. SEED = atoi(argv[4]);
3. //other code
4. initMatrix<int>(life, M, N, value);
5. //other code
6. printMatrix<int>(life, M, N);
7. //other code
8. std::cout << "No. of cells alive initially = ";
9. count = cellsAlive(life, M, N) ;
10.   std::cout << count << std::endl;
11.   // Play the game of life for given number of iterations
12.   t1=gettime();
13.   for (k = 0; k < NTIMES; k++) {
14.     // compute new matrix
15.     temp = compute(life, temp, M, N);
16.     // swap old and new matrices
17.     ptr = temp;
18.     temp = life;
19.     life = ptr;
20.   }
21.   t2= gettime();
22.   // Display the life matrix after NTIMES
23.   std::cout << "Life after " << NTIMES << " iterations:" <<
      std::endl ;
24.   printMatrix<int>(life, M, N);
25.   std::cout<< "No. of cells alive after " << NTIMES << "
      iterations = ";
26.   count = cellsAlive(life, M, N);
27.   std::cout << count << std::endl;
28.   //other code
```

Figure 4-16- Code snippet from the sequential game of life application

The steps to parallelize this application are expressed in the form of the Hi-PaL code shown in Figure 4-17. The line # 2 of Figure 4-17 expresses the intent that the two-dimensional integer-type matrix `life` with `M` rows and `N` columns that occurs in function ("`main`") should be distributed (or scattered) amongst different processors around the statement `initMatrix<int>(life, M, N, value);` (line # 4 of Figure 4-16). The code at line # 4 of Figure 4-17 means that the integer-type variable named `count` should be reduced (to collect the information about the number of cells that are alive in each block) and that this operation needs to take place at `multiple` places in function ("`main`")- which means, everywhere the specified search pattern is found in function main. If the keyword `multiple` is not specified as an argument to the API then the code for reduce

operation will be inserted at only one place, which is the first occurrence of the specified search pattern in function ("main"). Therefore, multiple can manipulate cross-cutting concerns within a single module. The code at line # 3 and line # 5 expresses the intent to exchange the updated values of the border-cells of the blocks of matrices life and temp.

```
1. Parallel section begins after ("SEED = atoi(argv[4]);") mapping
   is Linear{
2. ParDistribute2DArrayInt(life, M, N) around statement
     ("initMatrix<int>(life, M, N, value);") && in function
     ("main");
3. ParExchange2DArrayInt (life, M, N) before statement
     ("printMatrix<int>(life, M, N);") && in function ("main");
4. AllReduceSumInt(count, multiple) after statement
     ("count = cellsAlive(life, M, N);") && in function ("main");
5. ParExchange2DArrayInt (temp, M, N) before statement ("ptr =
     temp;") &&  in function ("main")
6. }
```

Figure 4-17- Hi-PaL code for parallelizing game of life application

A code snippet from the generated parallel application is shown in Figure 4-18. As noted from it, the code for reducing the count variable is inserted at two places - after line # 28 and line # 48 - because the keyword multiple was specified in the code at line # 4 of Figure 4-17 (**AllReduceSumInt**(count, multiple)). Because an **around** type of hook was specified at line # 2 of Figure 4-17, the statement that was specified as a search pattern in the hook definition (initMatrix<int>(life, M, N, value);) was deleted. The usage of **around** type of hook saved the extra time in initializing the life matrix in the generated code because it is going to be set to the block of data from the matrix life_Fraspa (see line # 21 of Figure 4-18) - this happens due to the distribute (or scatter) operation specified at Line # 2 of Figure 4-17.

```
1. //other code. Files included & declaration section extended.
2. SEED = atoi(argv[4]);
3. MPI_Init(NULL, NULL);
4. MPI_Comm_size(MPI_COMM_WORLD, &size_Fraspa);
5. MPI_Comm_rank(MPI_COMM_WORLD, &rank_Fraspa);
6. create_2dgrid(MPI_COMM_WORLD, &comm2d_Fraspa, &rowcomm_Fraspa,…);
7. create_diagcomm(MPI_COMM_WORLD, size_Fraspa, p_Fraspa,…);
8. rowmap_Fraspa.init(M, P_Fraspa, p_Fraspa);
9. colmap_Fraspa.init(N, Q_Fraspa, q_Fraspa);
10.   myrows_Fraspa = rowmap_Fraspa.getMyCount();
11.   mycols_Fraspa = colmap_Fraspa.getMyCount();
12.   M_Fraspa = M;
13.   N_Fraspa = N;
14.   M = myrows_Fraspa;
15.   N = mycols_Fraspa;
16.   //other code
17.   if (rank_Fraspa==0){
18.    life_Fraspa = allocMatrix<int>(life_Fraspa, M_Fraspa,
       N_Fraspa);
19.    initMatrix<int>(life_Fraspa, M_Fraspa, N_Fraspa, value);
20.   }
21.   life = split<int>(life_Fraspa, life, M_Fraspa, N_Fraspa,…);
22.   //other code
23.   life = exchange<int>(life, myrows_Fraspa + 2,…);
24.   printMatrix<int>(life, M, N);
25.   //other code
26.   std::cout << "No. of cells alive initially = ";
27.   count = cellsAlive(life, M, N) ;
28.   MPI_Allreduce(&count, &count_Fraspa, 1, MPI_INT,…);
29.   count = count_Fraspa;
30.   std::cout << count << std::endl;
31.   // Play the game of life for given number of iterations
32.   t1= MPI_Wtime();
33.   for (k = 0; k < NTIMES; k++) {
34.     // compute new matrix
35.     temp = compute(life, temp, M, N);
36.     temp = exchange<int>(temp, myrows_Fraspa + 2,…);
37.     // swap old and new matrices
38.     ptr = temp;
39.     temp = life;
40.     life = ptr;
41.   }
42.   t2= MPI_Wtime();
43.   // Display the life matrix after NTIMES
44.   std::cout << "Life after " << NTIMES << " iterations:" <<
      std::endl ;
45.   printMatrix<int>(life, M, N);
46.   std::cout<< "No. of cells alive after " << NTIMES << "
      iterations = ";
47.   count = cellsAlive(life, M, N);
48.   MPI_Allreduce(&count, &count_Fraspa, 1, MPI_INT,…);
49.   count = count_Fraspa;
50.   std::cout << count << std::endl;
//other code
```

Figure 4-18- Code snippet from the generated parallel game of life application

4.1.5 Image Processing

A test case from the image processing domain for performing the contrast operation [91] was considered. To perform the contrast operation, the image is read from a file and the pixel values are stored in an unsigned integer array. The root mean square (RMS) value of all the pixel values is calculated and this RMS value is then used to update the value of each pixel. The updated array is finally written to a file. The base code snippet of the sequential application is shown in Figure 4-19. In order to parallelize this application, the 1-dimensional array `masterbuf` should be distributed across the available processors and the results (the variable `mysum`) should be collected via reduce operation. The results of the contrast operation performed individually by the processors should be collected in another array via gather operation. The Hi-PaL code to express this intension is shown in Figure 4-20. The code snippet from the generated parallel code is shown in Figure 4-21. The MPI API for distributing, reducing and gathering the values are shown inserted at lines # 19, 21, and 29 of Figure 4-21.

```
1. //other code
2. t1 = gettime();
3. masterbuf = allocvector(masterbuf, N);
4. initvector(masterbuf, N, 0);
5. mysum = computerms(masterbuf, N);
6. //contrast operation
7. t2 = gettime();
8. //other code
```

Figure 4-19- Code snippet of the sequential image processing application

```
1. Parallel section begins before ("t1=gettime();") mapping is
   Linear {
2. ParDistribute1DArrayDouble(masterbuf, N) after statement
   ("initvector(masterbuf, N, 0);") && in function ("main");
3. ReduceSumDouble(mysum) after statement
4. ("mysum = computerms(masterbuf, N);") && in function ("main")
5. ParGather1DArrayDouble(masterbuf, N) before statement
   ("t2=gettime();") && in function ("main");
6. }
```

Figure 4-20- Code snippet of the Hi-PaL code for the image processing application

```
1. //other code
2. MPI_Init(&argc, &argv);
3. MPI_Comm_rank(MPI_COMM_WORLD, &rank_Fraspa);
4. MPI_Comm_size(MPI_COMM_WORLD, &size_Fraspa);
5. LinearMapping<int> mapping(N, size, 0, 1);
6. counts_Fraspa = mapping_Fraspa.getCounts();
7. displacements_Fraspa = mapping_Fraspa.getDisplacements();
8. mycount_Fraspa = counts_Fraspa [rank_Fraspa];
9. start_Fraspa = mapping_Fraspa.getStart();
10.   end_Fraspa = mapping_Fraspa.getEnd();
11.   N_Fraspa = N;
12.   N = mycount_Fraspa;
13.   if (rank_Fraspa == 0)
14.   {
         masterbuf_Fraspa = allocvector(masterbuf_Fraspa, …);
         initvector(masterbuf_Fraspa, N_Fraspa, …);
15.   }
16.   t1 = MPI_Wtime();
17.   masterbuf = allocvector(masterbuf, N);
18.   initvector(masterbuf, N, 0);
19.   MPI_Scatterv(masterbuf_Fraspa, counts_Fraspa,…);
20.   mysum = computerms(masterbuf, N);
21.   MPI_Reduce(&mysum, &mysum_Fraspa, 1, MPI_DOUBLE, MPI_SUM,…);
22.   mysum = mysum_Fraspa;
23.   N = N_Fraspa;
24.   if (rank == 0)
25.   {
26.      printf("\n Reduced Values is: %lf ", mysum);
27.   }
28.   //contrast operation
29.   MPI_Gatherv(masterbuf, mycount_Fraspa,…);
30.   t2 = MPI_Wtime();
31.   //other code
```

Figure 4-21- Code snippet of the generated parallel image processing application

4.1.6 Mandelbrot Set

The Mandelbrot Set is a commonly used example from the domain of complex dynamics and it involves fractals (objects that involve similar components at various scales). Generation of this set involves iteratively solving an equation of complex numbers. Any number belonging to the Mandelbrot Set is depicted in colors, whereas, the numbers that do not belong to the set are colored as white. The code snippet of the sequential version of the Mandelbrot Set generation application is shown in Figure 4-22, the Hi-PaL code for parallelizing the same is shown in Figure 4-23, and the generated

parallel code is shown on Figure 4-24. This test case involves the distribution and gathering of data in a 2-dimensional array and as can be noticed from Figure 4-24, the generated code has calls to function templates for splitting and gathering the data in the 2-dimensional array (line # 35 and 46).

```
1. //other code
2. N= 1000;
3. //other code
4. bigmat = allocarray(bigmat, M+2, N+2);
5. for(y=0; y < M+2; y++) {
6.   for(x=0; x < N+2; x++) {
7.     bigmat[y][x] = 55;
8.   }
9. }
10.   t1 = gettime();
11.   for(y=0; y < M+2; y++) {
12.     for(x=0; x < N+2; x++) {
13.    c.real = ((float) x - 500.0)/250.0;
14.    c.imag = ((float) y - 500.0)/250.0;
15.    color = compute(c, maxiter);
16.    bigmat[y][x]=color;
17.   }
18.   }
19.   t2=gettime();
```

Figure 4-22- Code snippet of the sequential Mandelbrot Set application

```
1. Parallel section begins after ("N= 1000;") mapping is Linear{
2. ParDistribute2DArrayInt(bigmat, M, N) before statement
   ("t1=gettime();") && in function ("main");
3. ParGather2DArrayInt(bigmat, M, N) after statement
   ("t2=gettime();") && in function ("main")
4. }
```

Figure 4-23- Hi-PaL Code for parallelizing the Mandelbrot Set

4.1.7 Genetic Algorithm for Content-Based Image Retrieval

Unlike the previous test cases, the test case presented in this section is a real world application. The Content-Based Image Retrieval (CBIR) technique is used to search images in large databases on the basis of the image content instead of the image captions [92]. The images are sliced into smaller semantic regions and are stored as blobs in the database. Each segment represents an individual semantic region of the original

image (*e.g.,* grass, tiger, and butterfly). The next step involves the extraction of features (color, texture, shape) for each image segment.

```
1. //other code
2. N= 1000;
3. MPI_Init(NULL, NULL);
4. MPI_Comm_size(MPI_COMM_WORLD, &size_Fraspa);
5. MPI_Comm_rank(MPI_COMM_WORLD, &rank_Fraspa);
6. create_2dgrid(MPI_COMM_WORLD, &comm2d_Fraspa,…);
7. create_diagcomm(MPI_COMM_WORLD, size_Fraspa, p_Fraspa,...);
8. rowmap_Fraspa.init(M, P_Fraspa, p_Fraspa);
9. colmap_Fraspa.init(N, Q_Fraspa, q_Fraspa);
10.   myrows_Fraspa = rowmap_Fraspa.getMyCount();
11.   mycols_Fraspa = colmap_Fraspa.getMyCount();
12.   M_Fraspa = M;
13.   N_Fraspa = N;
14.   M = myrows_Fraspa;
15.   N = mycols_Fraspa;
16.   if (argc != 2) {
17.    printf("Usage: %s <outputfile>\n", argv[0]);
18.    exit(-1);
19.   }
20.   if ((fp = fopen(argv[1],"w")) == NULL) {
21.    printf("Unable to open file %s for write\n", argv[1]);
22.    exit(-1);
23.   }
24.   bigmat = allocarray(bigmat, M+2, N+2);
25.   for(y=0; y < M+2; y++) {
26.    for(x=0; x < N+2; x++) {
27.     bigmat[y][x] = 55;
28.    }
29.   }
30.   if (rank_Fraspa == 0)
31.   {
32.    bigmat_Fraspa = allocMatrix<int>(bigmat_Fraspa,…);
33.    initMatrix<int>(bigmat_Fraspa, M_Fraspa, N_Fraspa, value);
34.   }
35.   bigmat = split<int>(bigmat_Fraspa, bigmat, M_Fraspa,…);
36.   t1 = MPI_Wtime();
37.   for(y=0; y < M+2; y++) {
38.    for(x=0; x < N+2; x++) {
39.     c.real = ((float) x - 500.0)/250.0;
40.     c.imag = ((float) y - 500.0)/250.0;
41.     color = compute(c, maxiter);
42.     bigmat[y][x]=color;
43.    }
44.   }
45.   t2= MPI_Wtime();
46.   bigmat_Fraspa = collect<int>(bigmat, bigmat_Fraspa,…);
47.   //other code
```

Figure 4-24- Code snippet of the generated Mandelbrot Set application

Because the amount of image data is large, clustering is used to preprocess the data and reduce the search space in the image retrieval process. The clustering is performed on image segments and therefore if a segment belongs to the cluster so does the image containing the segment. The clustering performed here is based on a Genetic Algorithm (GA). A typical experiment involved using 9,800 images with 82,556 regions and these image regions were divided into 100 clusters. Additional details and steps involved in the CBIR procedure can be found in [92].

In this experiment, if the GA is run for 100 generations or greater, it produces better quality of clusters. The MATLAB-based implementation of the GA took more than 4 hours to execute for 100 generations [92]. The MATLAB code was first converted to C/C++ code and then parallelized using MPI. The GA for CBIR is an excellent test case because it is computation-intensive. The code snippet of the sequential version of the GA is shown in Figure 4-25. Each generation of the GA involved 50 chromosomes, with 100 centroids on each chromosome. The centroids are the identification number associated with each image segment stored as a group of features, and as mentioned earlier, there are 82,556 image segments or regions involved in this experiment. The fitness value associated with each chromosome is the inverse of the sum of the minimum distances of each image segment from the centroids on each chromosome [92]. There are (82,556)*(100)*(50) computations involved in calculating the fitness values of the chromosomes in each population. The function in which the fitness value of the chromosome is calculated (`evaluatePop`) is therefore very time-consuming and application profiling showed that the application spends more than 90% of its execution time in this function. This function is therefore an ideal candidate for parallelization.

```
1. /*other code*/
2. evaluatePop(popcurrent,mydata,fitness);
3. for(i=0;i<numGenerations;i++){
4.   printf("Gen: %d ", i);
5.   pickchroms(fitness,popcurrent,popnext);
6.   mutation(popnext,popcurrent);
7.   equate(popcurrent, popnext);
8.   evaluatePop(popcurrent,mydata,fitness);
9.   printGenFit(popcurrent,fitness,(int)time1);
10.   }
11.   /*other code*/
```

Figure 4-25- Code snippet from the `main` function of sequential GA

The code snippet of `evaluatePop` function is shown in Figure 4-26. The for-loop on line # 4 of Figure 4-26, iterates over 82,556 image segments to compute the distance of each image segment from the centroid on each chromosome (`popcurrent[k][j]`). To split the task of the calculation of the distances amongst multiple processors, the computations in this for-loop should be split amongst multiple processors.

```
1. /*other code*/
2. for(k=0;k<numChrom;k++){
3.    sumDist=0.0;
4.    for(i=0;i<numofRecords;i++){
5.       min=maximVal;
6.       for(j=0;j<numCentroid;j++){
7.           z=popcurrent[k][j];
8.           eDist=0.0;
9.           for(l=0;l<=numVector;l++){
10.              eDist=eDist+(mydata[z][l]-mydata[i][l])*(…);
11.          }
12.          if(min>eDist){
13.              min=eDist;
14.          }
15.          min_d[i]=min;
16.      }
17.      sumDist=sumDist+sqrt(min_d[i])
18.   }
19.   fitness[k]= (1.00/sumTotal);
20.   }
21.   /*other code*/
```

Figure 4-26- Code snippet from the `evaluatePop` function in the sequential GA

The code snippet shown in Figure 4-27 shows the Hi-PaL code for parallelizing this for-loop which is outside the function `main` - (**`in function`** `("evaluatePop")`) and

collecting the results of the computations in the for-loop by reducing the value of `sumDist` before computing the fitness value. The code snippet of the parallelized `evaluatePop` function is shown in Figure 4-28. The MPI API for setting up the MPI-environment and other code for parallelization is inserted in the function `main` but is omitted here.

```
1. Parallel section begins before ("t1=gettime();") mapping is
   Linear{
2. Parallelize_For_Loop where (i=0;i<numOfrecords;i++)  before
   statement ("min=maximVal;") && in function ("evaluatePop");
3. ReduceSumInt(sumDist) before statement
   ("fitness[k]= (1.00/sumDist)") && in function ("evaluatePop")
4. }
```

Figure 4-27- Hi-PaL code for parallelizing the `evaluatePop` function in the GA

Because the GA can get stuck in local optima, it should be run for a large number of generations to obtain the globally optimal results. Therefore, it is imperative to checkpoint the application, especially when it is run in a dynamic and distributed environment. For checkpointing, depending upon the implementation scheme (type of load-balancing and design pattern) of the GA and the end-user's preference, the population and the fitness value of the chromosomes can be saved after certain number of generations or even during the last generation. The state of the executing GA application depends upon the current or initial population, and the seed value of the random-number generator function. For making this application fault-tolerant through checkpointing, the current population and the value of the seed used to initialize the random number generator function are stored in a file. The time of the day is passed as the seed value to the random number generator function in this application. To restart the program from any point in execution, the GA can be made to read the values of the seed of the random number generator and the current population from the restart files.

```
1. /*other code*/
2. long lower_limit_Fraspa;
3. long upper_limit_Fraspa;
4. int rank_Fraspa;
5. int size_Fraspa;
6. MPI_Comm_size(MPI_COMM_WORLD, &size_Fraspa);
7. MPI_Comm_rank(MPI_COMM_WORLD, &rank_Fraspa);
8. lower_limit_Fraspa=rank_Fraspa*((numOfrecords-1)…;
9. upper_limit_Fraspa =((rank_Fraspa==(size_Fraspa - 1))…)
10.   for(k=0;k<numChrom;k++){
11.      sumDist=0.0;
12.      for(i = lower_limit_Fraspa;i<=upper_limit_Fraspa;i++){
13.         min=maximVal;
14.         for(j=0;j<numCentroid;j++){
15.             z=popcurrent[k][j];
16.             eDist=0.0;
17.             for(l=0;l<=numVector;l++){
18.              eDist=eDist+(mydata[z][l]-mydata[i][l])*(…);
19.          }
20.          if(min>eDist){
21.              min=eDist;
22.          }
23.          min_d[i]=min;
24.      }
25.      sumDist=sumDist+sqrt(min_d[i])
26.   }
27.   MPI_Allreduce(&sumDist,&sumTotal,1,MPI_DOUBLE,MPI_SUM,…);
28.   fitness[k]= (1.00/sumTotal);
29.   }
30.   /*other code*/
```

Figure 4-28- Code snippet of the parallelized `evaluatePop` function in GA

A code snippet from the parallelized `main` function of GA is shown in Figure 4-29. The frequency of checkpointing, the CaR type (**`Centralized`**), and the loop variable `i` are specified in the DALC code in Figure 4-30, along with the name of the function `printGenFit after` whose `execution` the checkpointing code should be inserted. The restart mechanism is also specified through the DALC and is shown in Figure 4-31. Through this code, the `execution` of the function `dataInitialize` is intercepted. Due to this interception, instead of the execution of the initialization code in the function body, the array `popcurrent` is initialized with the values read from the file, `restartPopcurrent`. If the restart file is not present then the array is initialized using the values read from the file `initial`. The option of reading from one of these two files is

expressed by the usage of "|". The variable `time1`, which is passed as a seed to the random number generator, is initialized by the values read from the file `restartTime1`. The variables `numChrom` and `numCentroid` in Figures 4-30 and 4-31 are the dimensions of the array `popcurrent` and are provided by the user.

```
1. /*other code*/
2. for(i=0;i<numGenerations;i++){
3.    if (rank==0){
4.      printf("Generation #: %d",i);
5.    }
6.    popnext=pickchroms(fitness,popcurrent,popnext,start_x_y,…);
7.    MPI_Allgatherv(&popnext[0][0],(start_x_y.ystart_x_y.x)*…,…);
8.    if(rank==0){
9.      popcurrent=mutation(popcurrent,numOfrecords,…);
10.      }
11.   MPI_Bcast(&popcurrent[0][0],numChrom*numCentroid,…);
12.   evaluatePop(popcurrent,mydata,fitness,start_x_y2,…);
13.   printGenFit(popcurrent,fitness,(int)time1,i,rank);
14.   }
```

Figure 4-29- Code snippet of the parallelized `main` function of GA

The CaR mechanism described through the DALC, as shown in Figures 4-30 and 4-31, is translated into intermediate code that a PTE can understand to carry out the non-invasive transformation of the existing application into a checkpointed one. As per the specification, the PTE generates the base language code for file I/O. Two files, `restartTime1` and `restartPopcurrent` are opened and the value of the variable `time1` (which is the seed value) and the contents of the array `popcurrent` are saved to these files. The code snippet of the checkpointed code is shown in Figure 4-32. The inserted checkpointing code is at lines # 14-28 of Figure 4-32.

```
1. beginCheckpointing:
2. after execution("printGenFit")&& (frequency = 10)
   && (loopVar = "i" ) && (CaRType = Centralized){
3. SaveInt(time1,"restartTime1")
4. SaveIntArray2D(popcurrent, numChrom, numCentroid,
   "restartPopcurrent")
5. }
```

Figure 4-30- Checkpointing specifications for the GA

```
1. beginInitialization: around execution ("dataInitialize"){
2.     ReadIntVarFromFile(time1, "restartTime1")
3.     ReadIntArray2DFromFile(popcurrent, numChrom, numCentroid,
       "restartPopcurrent") |
4.     ReadIntArray2DFromFile(popcurrent, numChrom, numCentroid,
       "initial")
5. }
```

Figure 4-31- Restart specifications for the GA

```
1. /*other code*/
2. for(i=0;i<numGenerations;i++){
3.    if (rank==0){
4.      printf("Generation #: %d",i);
5.    }
6.    popnext=pickchroms(fitness,popcurrent,popnext,start_x_y,…);
7.    MPI_Allgatherv(&popnext[0][0],(start_x_y.ystart_x_y.x)*…,…);
8.    if(rank==0){
9.      popcurrent=mutation(popcurrent,numOfrecords,…);
10.   }
11.   MPI_Bcast(&popcurrent[0][0],numChrom*numCentroid,…);
12.   evaluatePop(popcurrent,mydata,fitness,start_x_y2,…);
13.   printGenFit(popcurrent,fitness,(int)time1,i,rank);
14.   if(rank==0){
15.     if (i % 10 == 0){
16.       newInputFile = fopen("restartPopcurrent", "w");
17.       storeVar = fopen("restartTime1", "w");
18.       fprintf(storeVar, "%d  ", time1);
19.       for (ii = 0; ii < numChrom; ii++){
20.         for (jj = 0; jj < numCentroid; jj++){
21.           fprintf(newInputFile, "%d  ", popcurrent[ii][jj]);
22.         }
23.         fprintf(newInputFile, "\n");
24.       }
25.       fclose(newInputFile);
26.       fclose(storeVar);
27.     }
28.   }
29.   }
```

Figure 4-32- Code snippet of the checkpointed parallel GA

4.2 Evaluation and Experimental Setup

All the experiments for this research were run on a 128 node dual-processor Xeon cluster (Olympus) in the Department of Computer and Information Sciences at the University of Alabama at Birmingham and the SGI Altix cluster at the Alabama Supercomputing Center. Each node in the Olympus cluster has 4 GB of RAM, low-

latency InfiniBand network, and 4 terabytes of disk space. The Altix cluster has 228 CPU cores, 1.5 terabytes of memory, and 10.8 terabytes of disk space. The FraSPA was evaluated according to the following criteria:

1. Performance and accuracy of the generated versions of the parallel code versus their manually-written counterparts.
2. Performance and accuracy of the generated versions of the checkpointed code versus their manually-written counterparts
3. The number of Lines of Code (LoC) that the programmer has to write in C/C++/MPI in order to manually parallelize a sequential application versus the number of lines of Hi-PaL code the programmer has to write for parallelizing the sequential application automatically.
4. The number of LoC that were generated by the framework in order to parallelize the applications.
5. The number of LoC reused for generating various applications.

4.3 Results and Analysis

The run-time and speedup of the manually-written parallel code was compared with the run-time and speedup of the code generated through the framework. The results are shown in Figures 4-33 to 4-39. A summary of the problem size and the execution time for different versions (sequential, manually-written parallel, and generated parallel) of all the test cases in presented in Table 4-2. Each application was run on different numbers of processors to test if they are scalable. No significant loss in performance was observed in any test case and the results from the generated version were almost identical

to that of the manually-written version. For all the test cases that we have considered till date, the performance of the generated application is within 5% of that of the manually-written application.

Table 4-2- Performance comparison of various test cases

Application	Problem Size	Number of Processor	Serial (in sec)	Parallel Manual (in sec)	Parallel Generated (in sec)
Prime Number	first 250,00,000 Numbers	30	92.30	9.25	9.31
Circuit Satisfiability	30 input bits	30	208.81	7.70	7.19
Poisson Solver	matrix size: 5000×5000, number of iterations: 5000	30	8391.41	984.49	985.06
Game of Life	matrix size: 5000×5000, number of iterations: 10,000	30	17056.8	622.86	628.03
Image Processing	image of size 10 million pixels	3	5.44	3.86	4.07
Mandelbrot Set	matrix of size: 10000×10000	10	3.26	0.52	0.52
Genetic Algorithm	number of chromosomes in a population: 50 number of centroids on each chromosome: 100 number of generations: 100 Number of image segments: 82,556	30	2505.86	236.92	237.02

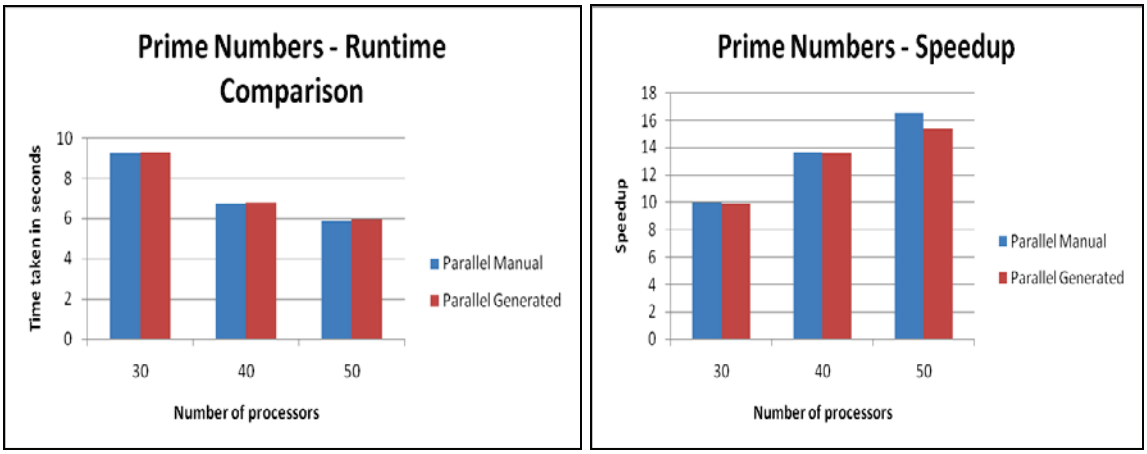

Figure 4-33- Runtime and Speedup – Prime Numbers

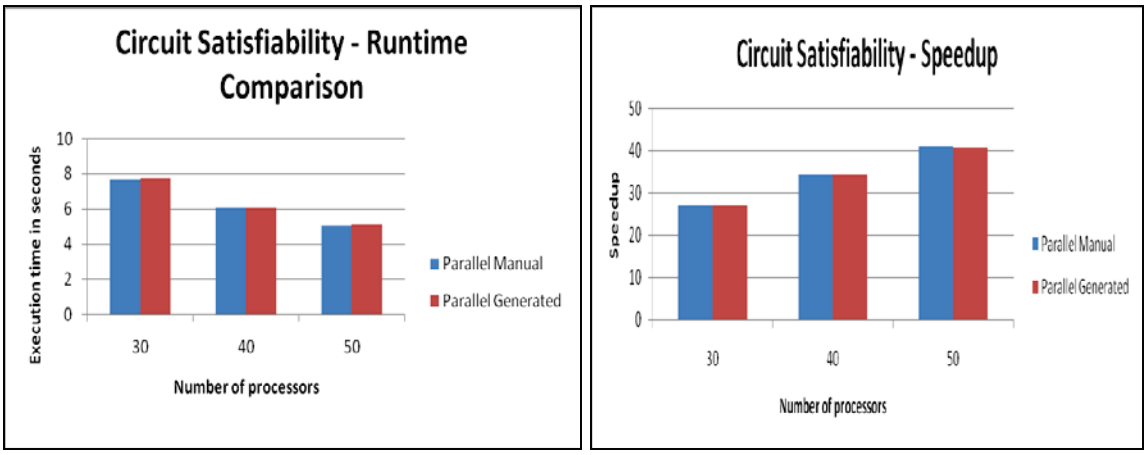

Figure 4-34- Runtime and Speedup – Circuit Satisfiability

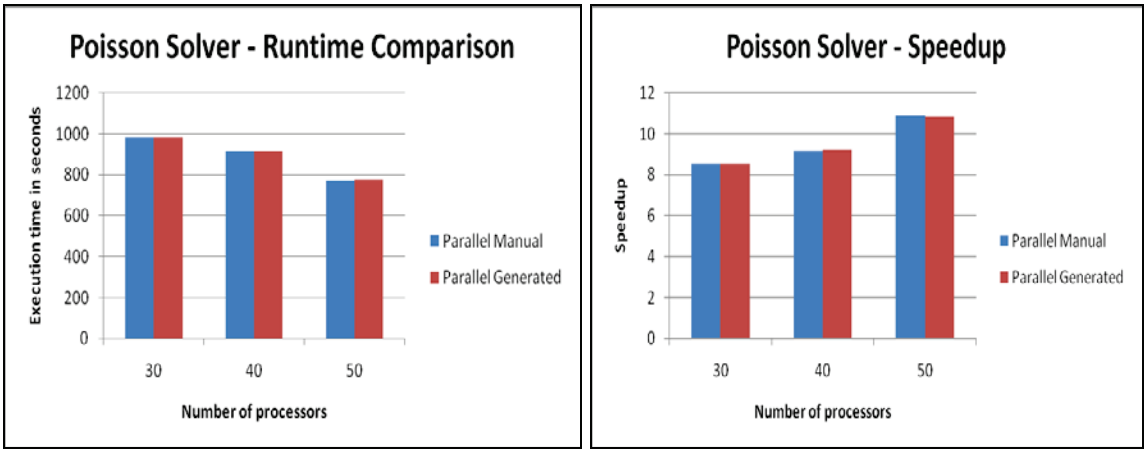

Figure 4-35- Runtime and Speedup – Poisson Solver

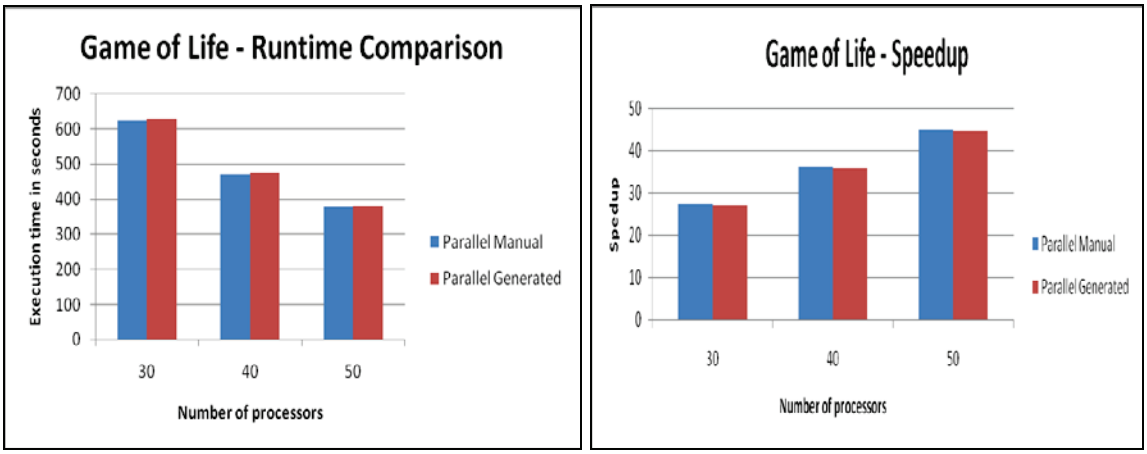

Figure 4-36- Runtime and Speedup – Game of Life

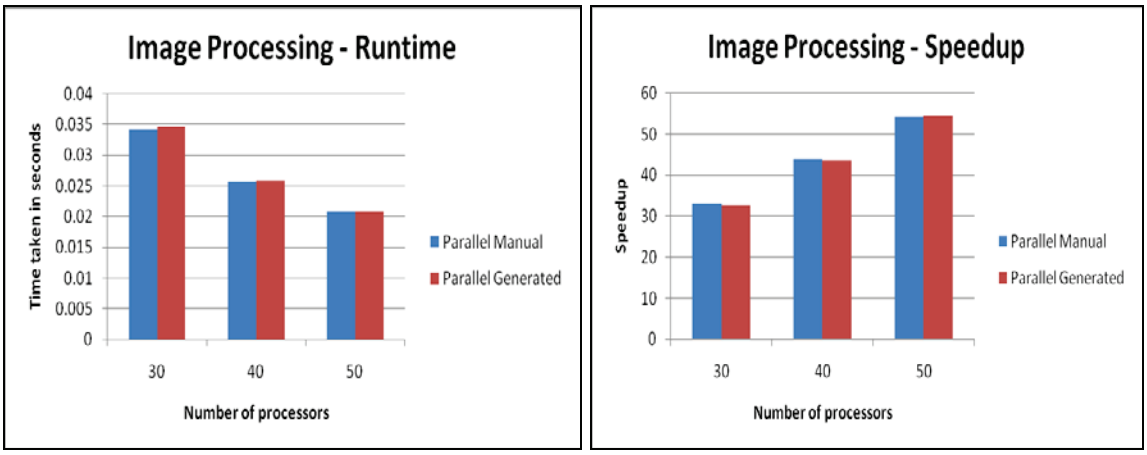

Figure 4-37- Runtime and Speedup – Image Processing

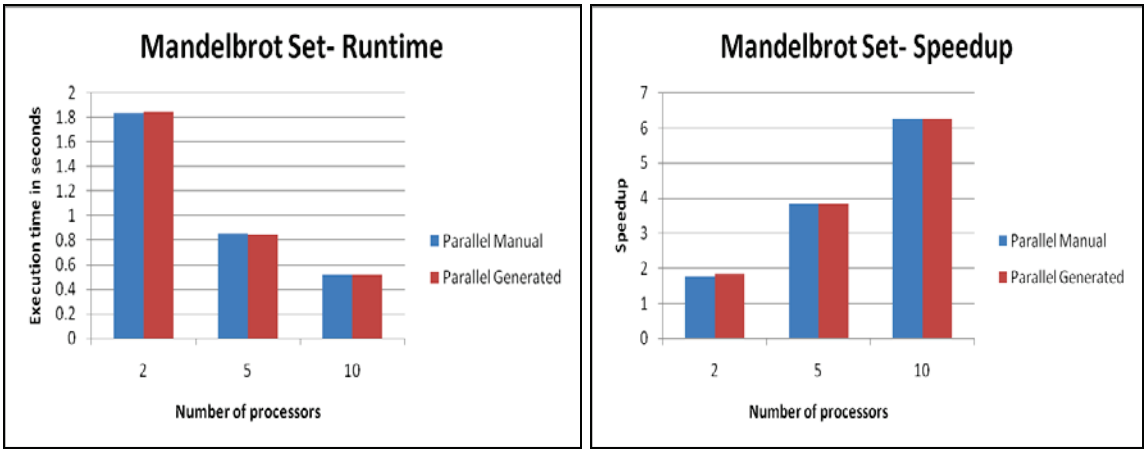

Figure 4-38- Runtime and Speedup – Mandelbrot Set

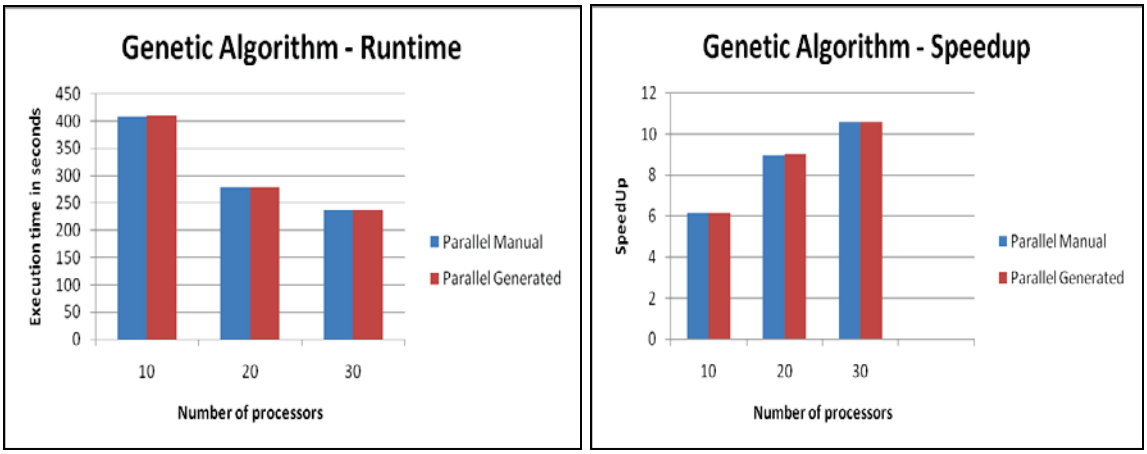

Figure 4-39- Runtime and Speedup – Genetic Algorithm

The results of checkpointing the test cases presented in Sections 4.1.2, 4.1.3 and 4.1.7 are presented in Figures 4-40 to 4-42. The results of checkpointing the Circuit Satisfiability application manually and through the DALC are presented in Figures 4-40. The application was run on 10 processors with 30 input bits. The total number of solutions that satisfied the circuit was 1920. The checkpointing was done every 10000, 20000, and 30000 iterations.

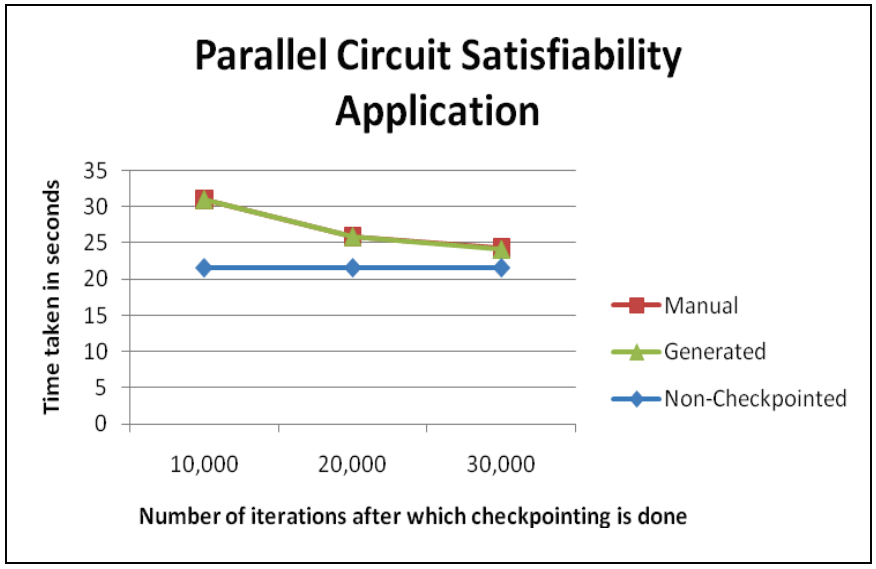

Figure 4-40- Runtime comparison of checkpointed Circuit Satisfiability application

The Poisson Solver application was also checkpointed both manually and through the DALC. The two versions of the application were run for 50,000 iterations for a 1000x1000 matrix. In both the versions, the convergence is reached after 41218 iterations. The application was run on 40 processors and the frequency of checkpointing was every 1000, 3000, and 5000 iterations. The comparison chart of the execution time is presented in Figure 4-41.

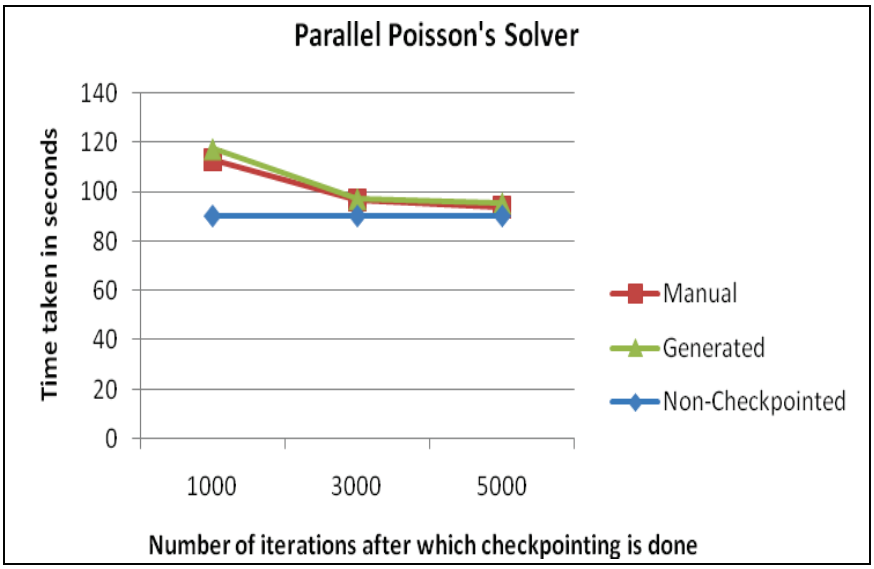

Figure 4-41- Runtime comparison of the checkpointed Poisson Solver application

The GA was run for 1000 generations on 50 processors. Because the execution time of the GA is very short, it was run for a greater number of iterations to study the impact of checkpointing. The checkpointing was done after every 10, 20 and 30 iterations. A comparison between the manual and the generated version of the checkpointed code of the GA is shown in Figure 4-42. The performance of the GA with the generated checkpointing code is comparable to the manually-checkpointed GA.

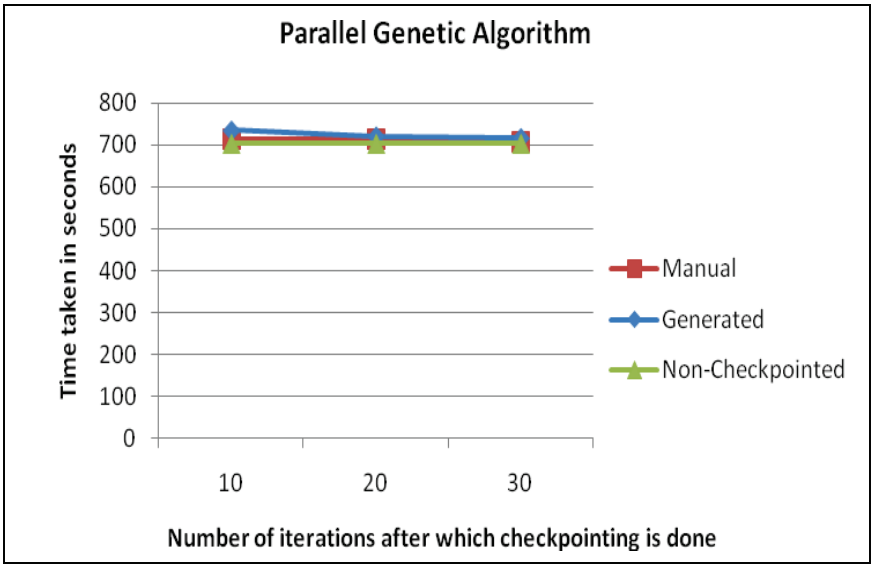

Figure 4-42- Runtime comparison of the checkpointed Genetic Algorithm

The performance of the version in which the CaR mechanism was generated through the DSL is within 5% of the version in which the CaR mechanism was inserted manually for all the test cases used in this research. The difference between the performance overheads of the generated and manually-written code (which is maximum 5% in the worst case) seems to be less apparent if the code is run for very large number of iterations and the checkpointing is done at a very low frequency. Though the GUI developed in this research can be used for generating the CaR-specifications (*i.e.,* the DSL code) in a wizard-driven manner, the end-user can also specify the DSL code manually. A snapshot of the GUI is shown in Figure 3-17. Hence, the end-user effort is reduced in terms of increase in code reuse.

A summary of the comparison of the number of LoC for the case-studies described in Section 4.1 is presented in Table 4-3. For example, for the Circuit Satisfiability application, the framework generates 104 LoC to parallelize the sequential version of the application. The generated code sets-up the parallel environment, terminates the MPI execution, has the necessary logic for carrying out the for-loop parallelization, and reducing the desired value. The programmer had to write just 4 lines of DSL code in order to parallelize this application. FraSPA uses various design templates for automatically generating the parallel code on the basis of the Hi-PaL code provided by the programmer. These templates are generic enough to be reused across applications from diverse domains. Some of the design templates that were reused across the test cases presented in Section 4.1 are the ones that set-up the MPI environment, data distribution, data collection, parallelization of for-loop, and exchanging the values across the cells in a stencil.

Table 4-3- Comparing the LoC for various test cases

Application	**Serial** (LoC)	**Parallel Manual**(LoC)	**Hi-PaL** (LoC)	**Parallel Generated**(LoC)
Prime Numbers	47	62	5	75
Circuit Satisfiability	86	102	4	104
Poisson Solver	85	330	6	357
Game of Life	180	623	6	641
Image Processing	59	150	4	159
Mandelbrot Set	93	494	3	515
Genetic Algorithm	431	816	3	830

On the basis of the results in Table 4-3, it can be observed that if Hi-PaL is used, there is a clear reduction in the programmer effort (more than 90%) in terms of the reduction in the number of lines of code that he or she has to write in order to accomplish the task of explicit parallelization using MPI. The programmer is also freed from the complexities associated with the process of explicit parallelization.

A summary of the number of LoC in the design templates that were reused to parallelize the test cases is presented in Table 4-4. As noted from Table 4-4, the reduce operation is performed twice in the parallel version of the application for finding the Prime Numbers. The design template for generating the code for the reduce operation has 335 LoC and it is called twice (`2*335`). The template for inserting the code for setting up the MPI environment has 452 LoC and it is reused across all the test cases. These 452 LoC are required to insert 7 significant lines of C/C++/MPI code (for setting up the MPI environment, extending the variable declaration section and including the required files) in the existing sequential applications. The template for the exchange operation has 263 LoC and it was used thrice in the Poisson Solver application. The exchange template, if used once in a program, internally invokes the distribution template and is responsible for

inserting 114 lines of significant C/C++/MPI code in the existing sequential application. With every additional invocation of the exchange template, an additional line of C/C++ code is inserted. Therefore, in total 116 lines of C/C++/MPI code are inserted by the invocation of the exchange templates in the Poisson Solver example.

The reusable code components (like design templates) are helpful in code maintenance and localization of changes. If any change is required in the implementation of the exchange operation for example, then the same is done at one place as compared to multiple places in multiple test cases. In summary, the code reuse through design templates reduces the scope of code duplication (as observed in Figure 1-2), helps in decreasing the effort involved in code maintenance, and promotes code correctness.

Table 4-4- Reusability metrics for some of the design templates for code generation

Application	**MPI Setup** (LoC)	**Reduce** (LoC)	**Distribution** (LoC)	**Exchange** (LoC)
Prime Numbers	452	2*(335)	0	0
Circuit Satisfiability	452	335	0	0
Poisson Solver	452	419	271	3*(263)
Game of Life	452	419	271	2*(263)
Image Processing	452	335	271	0
Mandelbrot Set	452	0	271	0
Genetic Algorithm	452	335	0	0

The total number of LoC required for implementing various components of FraSPA is shown in Figure 4-5. These statistics do not reflect the actual LoC written for evolving FraSPA into its current state. Before building the abstractions in FraSPA, the implementations of various transformation rules was done manually and the estimate of the same has not been provided here. There are approximately 14 KLOC in the current

deliverable. The front-end of FraSPA comprises of about 17% of the total LoC, the middle-layer consists of nearly 63% of the total LoC, and the backend is nearly 17% of the total LoC. The middle-layer, which is the thickest layer in the FraSPA architecture, captures the expertise required for parallelization and fault-tolerance in the form of reusable rule and design templates.

Table 4-5- Effort estimation in terms of LoC for developing FraSPA

Metamodels for DALC & Hi-PaL		
DALC Metamodel	Hi-PaL Metamodel	RSL Metamodel
Number of Classes in KM3 metamodel:175	Number of Classes in KM3 metamodel:510	Number of Classes in KM3 metamodel:172
LoC for TCS:370	LoC for TCS:674	LoC for TCS:345
ATL Rules		
Total Number of ATL Files: 16		
Total Number of LoC in ATL Files: 5752		
Ant Scripts		
Total Number of Scripts: 20		
Total Number of LoC in Scripts: 2294		
DMS Code (PARLANSE Functions)		
Total Number of LoC: 2401		
Total Number of Functions: 17		
C++ Design Templates		
Total Number of Templates: 7		
Total Number of LoC: 622		
Java Code		
Total Number of Files: 7		
Total Number of LoC: 364		

4.4 General Discussion and Summary

The parallelization and CaR code could also have been generated directly through the PTE without using the Hi-PaL or DALC. However, the time and complexity involved in learning and using the PTE necessitated a higher level of abstraction and the DSLs developed in this research provide the same. As noted from the results in Section 4.3, inserting the CaR mechanism and parallelization code using the DSL and the PTE is a

cost-effective option for non-invasive reengineering of large legacy applications to make them fault-tolerant and parallel. More than 90% reduction in the end-user effort was observed in the test cases that were generated through FraSPA and the performance of the generated code was within 5% of that of its manually-written counterpart. The problems related to maintaining different copies of the application are also overcome and it is easy to evolve the application. Because the original application does not undergo any restructuring, the readability and understandability of the legacy application is also maintained. The mechanism for fault-tolerance as developed in this research is platform-independent and can be used to checkpoint code written in several base languages with slight modifications made to the existing DALC [8].

The checkpointed application can be migrated from one resource to another without affecting the accuracy of the results [93]. If the resources are comparable, no significant loss in performance is observed. This DSL-based ALC-technique for making the parallel application fault-tolerant can be extremely useful in dynamic environments, like the grid, where small size of checkpoints and platform-independence are of prime importance. The fault-tolerance mechanism is not only useful in the scenario in which there is a possibility of resource failures but also for cost-effective resource scheduling.

With the current trend in the scientific community to adapt their applications for the cloud computing environment, cost-effective resource scheduling is of paramount importance. Imagine a scenario in which the computation nodes have a cost attached to them and their availability is not guaranteed. In order to develop an optimal scheduling strategy where the jobs get serviced at a reasonable cost and with a tolerable amount of delay, it will be imperative to move the jobs from one resource to another depending

upon the cost and availability ratios. The CaR mechanism developed to make the applications fault-tolerant can be used for developing the optimal scheduling strategy in which the jobs can be started on any resource that is available (if getting it serviced at the earliest is the priority) or it can be queued on the least expensive resource that is available (if the cost has to be kept lowest and the queue wait time is not of concern). However, apart from these two cases at the extreme ends, the middle-out approach (if the cost and service time are both of equal importance) would be to start the job on a particular resource, checkpoint it regularly, migrate it to a better resource as soon as it is available, and restart it from the latest checkpoint. The definition of a better resource as mentioned in the previous sentence is highly subjective and will depend upon the end-user preferences.

In the current implementation of the framework, the onus is entirely on the end-users to correctly identify the concurrency in their existing sequential applications and to be aware of the naming convention of the generated code. The programmers are required to manually ascertain that the operations inside the for-loop are independent of the results in the previous iterations and there are no data-dependencies in general in the code specified for parallelization. All the framework-generated variable names have a suffix **`_Fraspa`** and as per the guidelines provided to the end-users, they are expected to avoid naming their variables with this suffix to prevent name-conflicts between the generated and user-defined variables. This limitation can be removed in future by generating unique variable names after analyzing the existing code with the help of a static code analyzer.

It should also be noted that in the current implementation of FraSPA, the end-user is required to assure that all the partial computation results have been collected from the

processors and that the processors are in a synchronized state at the time of taking a distributed checkpoint.

CHAPTER 5

FUTURE WORK

This chapter outlines the directions in which the research presented in this dissertation can be further extended. By raising the level of abstraction of developing checkpointed (fault-tolerant) and MPI-based parallel applications through FraSPA, this dissertation solved two major challenges associated with the HPC application development. The HPC platforms can be broadly classified into two categories – homogeneous platforms and heterogeneous platforms. Within these categories, there are more categories depending upon the type of processing elements or memory-access pattern. FraSPA solved the problem of automating the process of generating parallel applications for distributed memory architectures.

FraSPA has the potential of being extended to support multiple parallel programming models (*e.g.,* support for synthesizing parallel applications for shared memory paradigms and multi-core architectures) and hence multiple parallel programming platforms. In order to extend FraSPA to develop applications for shared memory architectures, a new DSL should be developed instead of extending Hi-PaL. The mapping between the DSL in the front-end and the program transformation engine in the backend will need to be extended too. However the backend will not undergo any changes. The most tedious aspect of writing an OpenMP program is identifying the variables that are meant to be kept private or shared amongst the multiple threads that

work in parallel on a section of a program. This part can be made semi-automatic by using a static code analysis tool and a GUI. A summary of the suggestions regarding which variable could be made private or shared and the directives to use can be presented to the end-user to make selections from. As compared to MPI API, OpenMP has a lesser number of directives for parallelization and the main ones are for the parallelization of loops and the reduction operation. Therefore, compared to Hi-PaL, the DSL developed for explicit parallelization using OpenMP will be simpler. The DALC can make C/C++/OpenMP-based applications checkpointed without undergoing any changes. It would also be of interest to extend FraSPA to provide the functionality of the framework developed for raising the level of abstraction of GPGPU programming [94].

In addition to providing support for multiple programming paradigms, FraSPA can also be extended to support the automatic parallelization of sequential applications written in other legacy languages (*e.g.,* FORTRAN) and dialects. For supporting more legacy languages and dialects, the back-end support should be extended without much change required in the front-end (*i.e.,* Hi-PaL or DALC). Further details for providing support for transforming code written in other legacy languages are provided in [37, 81].

Besides providing support for transforming legacy applications, FraSPA can also be extended to support development of new applications in implicitly parallel languages like SISAL and X10. To achieve this goal, design-templates can be developed to capture the known patterns of writing the code in these languages. FraSPA can use these design-templates to generate a stub-and-skeleton type of code template in which the end-user will only need to provide the computation kernel in a high-level language [60, 63, 73].

It is crucial to include the mechanisms for supporting fault-tolerance, resource-selection, and feedback for improved performance in a framework that would support the development of applications for complex heterogeneous systems [7, 69, 75]. In this context, DALC, and hence FraSPA, can make C/C++ programs fault-tolerant without any extensions or modifications. However, to make the applications running on GPGPU checkpointed (and hence fault-tolerant), more work is required for capturing the domain-knowledge in terms of additional classes in the DALC metamodel. FraSPA can be extended to generate resource-aware parallel applications that can run in grid computing environments [80] by providing support for automatic resource discovery and adaptation. In the grid computing environment, the resources can be heterogeneous, dynamic and distributed. In order to dynamically and automatically generate a parallel application for a heterogeneous platform from a set of Hi-PaL specifications and a sequential application, a repository of application-characteristics is required [73, 95]. This repository will contain the information of the performance of a class of application on a particular resource (*e.g.,* performance of evolutionary algorithms on multi-core architecture). Through the repository of the application-characteristics, a sorted list of the resources the application can be run on could be generated. The probability of the resource availability can also be determined on the basis of the historical-data. An optimization function to select the resource (HPC platform) that is most likely to be available and that has the best application performance can be designed. Depending upon the dynamically selected resource (*e.g.,* shared memory platform and distributed platform), FraSPA should be able to automatically generate an optimized and checkpointed parallel application from the Hi-PaL specifications and the sequential application.

A feedback mechanism will be especially useful in the heterogeneous environments for selecting the best algorithm or implementation scheme from the given solution space of platform-specific implementations [7]. If the applications do not perform optimally on the architecture selected by the end-user or FraSPA, the information about the same could be fed into FraSPA and it can be trained to improve the generated solutions. This mechanism will also be helpful in updating the repository of application-characteristics.

An application-profiler (like Vtune from Intel [95]) can also be integrated into the FraSPA framework such that if the end-user is not satisfied with the performance of the generated code, they can further identify the hotspots for parallelization and fine-tune the parallel application. This process of fine-tuning can also become a part of the feedback mechanism discussed in the previous paragraph. Currently FraSPA does not have any facility to prompt the end-user if they are selecting a wrong combination of parallel operations (*e.g.,* gather operation a variable instead of reduce operation). If the programmer does not specify the correct parallel task, FraSPA will still generate the code for parallelization as long as it finds the match-pattern and other constraints are satisfied.

The current set of guidelines developed for parallelizing the sequential code is coarse-grained. Therefore, a code-analyzer could be provided along with the application-profiler so that the programmer can not only detect the hotspots for parallelization but can also make informed choices about the parallel operations to choose. For example, it might be hard for the programmer to find the dependencies in the for-loop manually. Therefore, a code analyzer can be helpful in this scenario for warning the programmer to avoid parallelizing a for-loop with dependencies.

Apart from the aforementioned extensions to FraSPA, the following are some of the potential areas of enhancements:

- The current implementation of the framework supports a limited set of C++ grammar rules for the VisualC++ 6.0 dialect. Extra effort is required to provide support for the complete C++ grammar for all the dialects in order to make the framework useful for transforming applications on a large-scale. A richer support for specifying complex hooks (or join points) is required (*e.g.,* run-time evaluation of control-flow).
- The possibility of adopting the memory hierarchy aware algorithm design approach [96, 97] for improving the existing design-templates used in FraSPA could be explored in future. For example, code can be structured to maximize locality and tasks can be parameterized (multiple implementations of a particular task can be provided) in order to produce highly optimized parallel code [7].
- The process of making the applications fault-tolerant via checkpointing has already been made wizard-driven as a part of this research [22]. Efforts could be made to make the process of specifying the Hi-PaL code wizard-driven as well.
- The facility to search the join point or hook in the sequential application on the basis of the logical line number can also be provided to reduce the effort in specifying the match-pattern.

CHAPTER 6

SUMMARY AND CONCLUSION

The combination of emerging computing platforms (like GPUs, cell processors, and field-programmable gate arrays) with traditional CPUs is being publicized as the next revolution in HPC. Though such a paradigm shift is bound to bring massive amount of computational power to the end-user at a low-cost, there is no unified domain-neutral application development environment at the time of writing this dissertation that allows the end-user to express concurrency at a high-level and dynamically generate optimal solutions for even homogeneous HPC platforms. There are multiple parallel programming paradigms, each best-suited for developing applications for a specific computing platform. Therefore, the end-users (or domain-experts) are stuck in the "problem of plenty" and experience a steep learning curve with each HPC platform or programming model. Due to the increasing diversity in the type of processing elements in the modern HPC platforms and the drastic increase in the number of processing elements on a chip, the probability of failures of the processing elements is also increasing thereby leading to reduced MTBF [5]. Therefore, a mechanism for supporting fault-tolerance is required to make the applications running on such platforms immune to resource-failures.

In the light of the aforementioned changes in the HPC landscape, the goal of this dissertation was to take the first step towards developing a framework that brings scalability and performance to the end-user in the form of parallel computing without the

need to learn any low-level parallel programming paradigm or to do any manual intrusive reengineering. A framework, named FraSPA, was developed in this research and in its current scope, it supports the generation of checkpointed and MPI-based parallel applications from the existing applications (written in C/C++) and code components on the basis of the high-level specifications provided by the end-user.

Two DSLs have been developed as a part of this research for obtaining the high-level specifications form the end-user and they are called Hi-PaL and DALC. Both DSLs were developed from scratch and borrow some concepts from the AOP techniques. With the help of Hi-PaL, without knowing anything about MPI API or its usage, end-users can specify the tasks required for parallelizing the existing sequential applications at a very high-level. The end-users are, however, expected to be familiar with the logic of the sequential application and should be well acquainted with the concept of concurrency. A set of Hi-PaL API has been developed for the commonly used parallel tasks like data distribution, data collection, reading or writing the data in parallel, and parallelizing a for-loop. DALC is useful for obtaining the specifications for checkpointing and restart from the end-users. It includes the API for periodically saving and reading the critical application variables from which the complete execution state of an application can be recreated in the event of a failure of underlying resources.

Hi-PaL and DALC act as an interface between the end-user and FraSPA. The specifications provided by the end-user (in the form of Hi-PaL or DALC code) are translated into the rules for the source-to-source compiler at the back-end by the Rule Generator. On the basis of these rules, the source-to-source compiler instruments the existing application to make it checkpointed or parallel without requiring any manual-

reengineering. The code generation process is generic and domain-neutral due to the domain-knowledge that is captured in the Rule Generator component of FraSPA. The Rule Generator not only contains the domain-knowledge for generating the appropriate rules from the Hi-PaL and DALC code, but also invokes the source-to-source compiler (which is DMS in this research), handles the input to DMS, and gets the output from the DMS to the end-user workspace.

With FraSPA, the programmer is not required to adapt the application to any generic interfaces, can do incremental integration of components, and need not restructure the existing application. The test cases presented in Chapter 4 of the dissertation demonstrate the usage of FraSPA for doing various automatic transformations (*e.g.,* manipulating the declaration section, including files, inserting a library call and deleting a line of code) and demonstrate its domain-neutral nature. FraSPA demonstrates the desired flexibility to experiment with multiple communication patterns and algorithms. For the selected test cases, there is more than 90% of reduction in the end-user effort in terms of the number of lines of code written manually while requiring no explicit changes to the existing code. The performance of the generated code is within 5% of that of the manually-written code. FraSPA supports separation of concerns and thereby aids in code maintenance and evolution.

The processes of parallelizing an application and making it fault-tolerant via checkpointing are decoupled from each other in FraSPA –*i.e.,* they are two different steps. This gives the end-user a choice of using the two mechanisms separately. If the end-user is interested in parallelizing their sequential application, they can do so by using Hi-PaL. If they are interested in making their parallel application fault-tolerant by

inserting the checkpointing and restart mechanism, irrespective of the fact whether the application was generated by FraSPA or manually-written, they can do that as well by using DALC. Both these steps obviate the need to intrusively reengineer the existing application to make it parallel and/or checkpointed. The complex workflow inside FraSPA and the complexities associated with the process of explicit parallelization are hidden from the end-user.

The evaluation metrics presented in Chapter 4 show that FraSPA not only raises the level of abstraction of non-invasively generating checkpointed parallel programming without drastically degrading the performance, but it also promotes code reusability. Because FraSPA supports separation of concerns, the process of developing HPC-applications can become a multi-person software development activity. The domain-experts can focus on developing the sequential parts of the application or providing the specifications for parallelization through Hi-PaL and the computer scientists can work on developing the optimized code components for parallelization and fault-tolerance that can be integrated into FraSPA for improving the performance of the generated code.

Because the source-to-source compiler used in this research, DMS, is robust and capable of handling large-scale applications, scalability of the approach presented in this research is not an issue. This research not only shows a high-level technique for synthesizing fault-tolerant parallel applications, but also shows a mechanism for raising the level of abstraction of the DMS usage (the accidental complexities associated with the usage of DMS are explained in Chapter 3).

FraSPA has the potential of being extended to support multiple programming languages and paradigms such that eventually the problem of the lack of a unified

software development for heterogeneous platforms can be solved. The limitations of FraSPA are discussed in Chapter 4 and the various measures to improve on them are presented in Chapter 5. Code snippets of some of the components of FraSPA are presented in the Appendices. Before concluding this dissertation, it is worth revisiting the questions raised in Section 1.1.4 of Chapter 1 and to ponder if this research helps in answering any of those. The questions were as follows:

1. Is it feasible to achieve portability and optimal performance with reasonable effort?
2. Can efficient parallel programs be automatically generated by computers?
3. Can we bring scalability and performance to domain-experts in the form of parallel computing without any need to learn low-level parallel programming?
4. Can we facilitate the transition of HPC from the realms of specialized and scientific application development into mainstream business?
5. Can we mitigate the negative impact of the reduced MTBF of the complex parallel computing platforms on the execution time of the applications?

While the answer to questions 2, 3, and 5 is a clear “yes”, more work is required to find a conclusive answer to questions 1 and 4. Because FraSPA is capable of automatically generating performance-oriented and checkpointed parallel programs on the basis of the end-user specifications, it answers questions 2, 3, and 5 in the affirmative. Though the applications generated by FraSPA show a drastic reduction in end-user effort (more than 90%) in terms of the number of lines of code written manually, and the performance of the generated code is commensurate to the effort spend (within 5% of that of the manually-written code), more work is required to extend FraSPA to support

heterogeneous architectures. Therefore, the answer to question 1 is a “partial yes” at the time of writing this dissertation. Because FraSPA can mitigate the complexities associated with low-level parallel programming, it has the potential to lower the barriers to large-scale HPC adoption [1, 2]. However, usability-studies are required to test this feature of FraSPA and hence the answer to question 4 is yet to be discovered.

LIST OF REFERENCES

[1] Earl Joseph, Christopher G. Willard, Dolores Shaffer, Addison Snell, Suzy Tichenor, Steve Conway, "Council on Competitiveness Study of ISVs Serving the High Performance Computing Market. Part A – Current Market Dynamics": http://www.compete.org/images/uploads/File/PDF%20Files/ISV_Study_Part_A_2005.pdf

[2] High Performance Computing Reveal, "Council on Competitiveness and USC-ISI Broad Study of Desktop Technical Computing End Users and HPC": http://www.compete.org/images/uploads/File/PDF%20Files/CoC_REVEAL_May19.pdf

[3] William Gropp, Ewing Lusk, Anthony Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 1999, pp. 1-371.

[4] The OpenMP API specification for parallel programming: http://openmp.org/wp/

[5] Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams, Katherine A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," *Electrical Engineering and Computer Sciences, University of California at Berkeley*, Tech. Rep. UCB/EECS-2006-183, December 2006. Available at: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf

[6] Top 500 Supercomputer Sites: http://www.top500.org/

[7] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, Nicholas Rizzolo, "SPIRAL: Code Generation for DSP Transforms," *IEEE*, *Special issue on Program Generation, Optimization, and Adaptation*, Vol. 93, No. 2, 2005, pp. 232- 275.

[8] Ritu Arora, Purushotham Bangalore, Marjan Mernik, "A technique for non-invasive application-level checkpointing," *The Journal of Supercomputing*, ISSN: 0920-8542, 2010, pp. 1-29.

[9] Victor Basili, Jeff Carver, Daniela Cruzes, Lorin M. Hochstein, Jeff Hollingsworth, Forrest Shull, Marvin Zelkowitz, "Understanding the High Performance Computing

Community: A Software Engineer's Perspective," *IEEE Software*, Vol. 25, No. 4, 2008, pp. 29-36.

[10] Frederick P. Brooks, Jr., "No Silver Bullet Essence and Accidents of Software Engineering," *Computer*, Vol. 20, No. 4, 1987, pp.10-19.

[11] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, Vivek Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *OOPSLA 2005*, pp. 519-538.

[12] Henri E. Bal, M. Frans Kaashoek, Andrew S. Tanenbaum, "Orca: A Language for Parallel Programming of Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. 18, No. 3, 1992, pp. 190-205.

[13] John T. Feo, David C. Cann, Rodney R. Oldehoeft, "A report on the Sisal language project," *Journal of Parallel and Distributed Computing*, Vol. 10 No. 4, 1990, pp. 349-366.

[14] Guy Steele, "Parallel programming and parallel abstractions in fortress," *Functional and Logic Programming, 8th International Symposium* (FLOPS 2006), LNCS 3945, 2006, pp. 1.

[15] Vincent W. Freeh. "A comparison of implicit and explicit parallel programming," *Journal of Parallel and Distributed Computing*, Vol. 34, No. 1, 1996, pp. 50-65.

[16] Ritu Arora, Purushotham Bangalore, "A framework for raising the level of abstraction of explicit parallelization," *International Conference on Software Engineering (ICSE)* Companion 2009, pp. 339-342.

[17] Anthony Skjellum, Purushotham Bangalore, Jeff Gray, and Barrett Bryant, "Reinventing Explicit Parallel Programming for Improved Engineering of High Performance Computing Software," *ICSE 2004 Workshop: International Workshop on Software Engineering for High Performance Computing System Applications*, 2004.

[18] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin, "Aspect-Oriented Programming," *European Conference on Object-Oriented Programming*, Springer-Verlag LNCS 1241, 1997, pp. 220-242.

[19] Krzysztof Czarnecki and Ulrich Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000, pp. 1-832.

[20] Marjan Mernik, Jan Heering, Anthony M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys*, Vol. 37, No. 4, 2005, pp. 316-344.

[21] Douglas Schmidt, "Guest Editor's Introduction: Model-Driven Engineering," *IEEE Computer*, Vol. 39, No. 2, February 2006, pp. 25-31.

[22] Ritu Arora, Purushotham Bangalore, Marjan Mernik, Suman Roychoudhury, Saraswathi Mukkai, "A Domain-Specific Language for Application-Level Checkpointing," *International Conference on Distributed Computing and Internet Technology*, 2008, pp. 26-38.

[23] Uwe Aßmann, *Invasive Software Composition*, Springer, 2003.

[24] AspectC++: http://www.aspectc.org/

[25] AspectC: http://www.cs.ubc.ca/labs/spl/projects/aspectc.html

[26] Purushotham Bangalore, "Generating parallel applications for distributed memory systems using aspects, components, and patterns," *6th Workshop on Aspects, Components, and Patterns For infrastructure Software* (ACP4IS '07), Vol. 219, 2007.

[27] Ritu Arora, Purushotham Bangalore, "Using Aspect-Oriented Programming for Checkpointing a Parallel Application," *Parallel and Distributed Processing Techniques and Applications* (PDPTA 2008), pp. 955-961.

[28] Bruno Harbulot and John Gurd, "Using AspectJ to Separate Concerns in a Parallel Scientific Java Code," *International Conference on Aspect-Oriented Software Development*, 2004, pp. 122-131.

[29] Bruno Harbulot and John Gurd, "A Join Point for Loops in AspectJ," *Workshop on Foundations of Aspect-Oriented Languages*, 2005.

[30] Mikhail Chalabine and Christoph Kessler, "Crosscutting Concerns in Parallelization by Invasive Software Composition and Aspect Weaving," *39th Hawaii International Conference on System Sciences*, 2006.

[31] Reuseware Composition Framework: http://www.reuseware.org/index.php/Main_Page

[32] Andreas Leha, Mikhail Chalabine, Christoph Kessler, "Parallelizing Scientific Code with Invasive Interactive Parallelization - A Case Study with Reuseware," *Int. Workshop on Component-Based High Performance Computing (CBHPC-2008),* 2008.

[33] Ira Baxter, "Design Maintenance Systems," *Communications of the ACM*, Vol. 35, No. 4, April 1992, pp. 73-89.

[34] Ira Baxter, Christopher Pidgeon, and Michael Mehlich, "DMS: Program Transformation for Practical Scalable Software Evolution," *International Conference on Software Engineering*, 2004, pp. 350-354.

[35] ROSE homepage: http://www.rosecompiler.org/

[36] David Wile, "Lessons Learned from Real DSL Experiments," *Science of Computer Programming*, Vol. 51, No. 3, 2004, pp. 265-290.

[37] Suman Roychoudhury, Frédéric Jouault and Jeff Gray, "Model-Based Aspect Weaver Construction," *International Workshop on Language Engineering*, 2007, pp. 117-126.

[38] The AMMA Platform: http://atlanmod.emn.fr/AMMAROOT/

[39] Jean Bézivin, Frédéric Jouault, David Touzet, "Principles, Standards and Tools for Model Engineering," *International Conference on Engineering of Complex Computer Systems*, 2005, pp. 28-29.

[40] Frédéric Jouault and Jean Bézivin, "KM3: a DSL for Metamodel Specification," *Formal Methods for Open Object-Based Distributed Systems*, Springer-Verlag LNCS 4037, 2006, pp. 171-185.

[41] Eclipse Modeling Framework Project: http://www.eclipse.org/modeling/emf/?project=emf

[42] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev, "TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering," *Generative Programming and Component Engineering*, 2006, pp. 249-254.

[43] Frédéric Jouault and Ivan Kurtev, "Transforming Models with ATL," *Model Transformations in Practice Workshop at MoDELS*, 2005.

[44] *OMG: Object Constraint Language Specification, version 2.0*, formal/2006-05-01, http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf, 2001.

[45] Jean Bézivin, Frédéric Jouault, Peter Rosenthal, Patrick Valduriez, "Modeling in the Large and Modeling in the Small," *MDAFA'2004*, Springer-Verlag LNCS 3599, pp. 33-46.

[46] Ivan Kurtev, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez, "Model-based DSL Frameworks," *Object-Oriented Programming, Systems, Languages and Applications Companion*, pp. 602-616.

[47] Rhiju Das, Bin Qian, Srivatsan Raman, Robert Vernon, James Thompson, Philips Bradley, Sagar Khare, Micheal D. Tyka, Divya Bhat, Dylan Chivian, David E. Kim,

William H. Sheffler, Lars Malmström, Andrews M. Wollacott, Chu Wang, Ingemar Andre, David Baker, "Structure prediction for CASP7 targets using extensive all-atom refinement with Rosetta@home," *Proteins*, Vol. 69, No. S8, 2007, pp. 118-128.

[48] Qingshan Chen, Jacques Laminie, Antoine Rousseau, Roger Temam, Joseph J. Tribbia, "A 2.5 model for the equations of the ocean and the atmosphere," *Analysis and Applications*, Vol. 5, No. 3, 2007, pp. 199-229.

[49] Milos Prvulovic, Josep Torrellas, Zheng Zhang, "Revive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors," *International Symposium on Computer Architecture*, 2002, pp. 111-122.

[50] Jason Duell, "The Design and Implementation of Berkeley Lab's Linux Checkpoint/ Restart," *Lawrence Berkeley National Laboratory, Paper LBNL-54941*, 2005, http://crd.lbl.gov/~jcduell/papers/blcr.pdf.

[51] Michael Litzkow, Todd Tannenbaum, Jim Basney, Miron Livny, "Checkpoint and Migration of Unix Processes in the Condor Distributed Processing System," *University of Wisconsin-Madison Computer Science Technical Report #1346* 1997.

[52] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Paul Stodghill, "Automated Application-Level Checkpointing of MPI Programs," *Symposium on Principles and Practice of Parallel Programming* (PPOPP 2003), pp. 84-94.

[53] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, Martin Schulz, "Application-level checkpointing for shared memory programs," *Architectural Support for Programming Languages and Operating Systems* (ASPLOS 2004), pp. 235-247.

[54] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Radu Rugina, "Compiler-Enhanced Incremental Checkpointing," *Languages and Compilers for Parallel Computing*, 20th International Workshop, LCPC 2007, pp. 1-15.

[55] Joshua Haines, Vijay Lakamraju, Israel Koren, C. Mani Krishna, "Application-Level Fault Tolerance as a Complement to System-Level Fault Tolerance," *The Journal of Supercomputing*, Vol. 16, Nos. 1-2, 2000, pp. 53-68.

[56] John Paul Walters, Vipin Chaudhary, "Application-Level Checkpointing Techniques for Parallel Programs," *International Conference on Distributed Computing and Internet Technologies* (ICDCIT 2006), pp. 221-234.

[57] Ajit Singh, Jonathan Schaeffer, Duane Szafron, "Experience with parallel programming using code templates," *Concurrency: Practice and Experience*, Vol. 10, 1998, pp. 91-120.

[58] Stephen Siu, Ajit Singh, "Design Patterns for Parallel Computing Using a Network of Processors," *International Symposium on High Performance Distributed Computing* (HPDC'97), 1997, pp. 293-304.

[59] Narjit Chadha, "A Java Implemented Design-Pattern-Based System for Parallel Programming," *Master of Science thesis, University of Mannitoba*, 2002.

[60] Dhrubajyoti Goswami, Ajit Singh, Bruno R. Preiss, "Building Parallel Applications using Design Patterns," *Advances in Software Engineering: Topics in Comprehension, Evolution and Evaluation*, Springer-Verlag 2002, pp. 243-265.

[61] Mikhail Chalabine, Christoph Kessle, "Parallelisation of sequential programs by invasive composition and aspect weaving," *6th International Workshop on Advanced Parallel Processing Technologies* (APPT'05). LNCS, 2005, pp. 131–140.

[62] João L. Sobral, "Incrementally Developing Parallel Applications with AspectJ," *20th IEEE International Parallel & Distributed Processing Symposium* (IPDPS'06), 2006.

[63] Fethi A. Rabhi, Helen Cai, Brian C. Tompsett, "A Skeleton-Based Approach for the Design and Implementation of Distributed Virtual Environments," *5th International Symposium on Software Engineering for Parallel and Distributed Systems*, IEEE Computer Society Press, 2000, pp. 13-20.

[64] Tarek El-Ghazawi, Francois Cantonnet, "UPC performance and potential: a NPB experimental study," *ACM/IEEE conference on Supercomputing*, 2002, pp. 1-26.

[65] Cristian Coarfa, Yuri Dotsenko, Jason Eckhardt, John Mellor-Crummey, "Co-array Fortran Performance and Potential: An NPB Experimental Study," *The 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003)*, 2003, pp. 177-193.

[66] Bradford L. Chamberlain, David Callahan, Hans P. Zima, "Parallel Programmability and the Chapel Language," *International Journal of High Performance Computing Applications*, Vol. 21, No. 3, 2007, pp. 291-312.

[67] Katherine Yelick, Paul Hilfinger, Susan Graham, Dan Bonachea, Jimmy Su, Amir Kamil, Kaushik Datta, Philip Colella, Tong Wen, "Parallel Languages and Compilers: Perspective From the Titanium Experience," *International Journal of High Performance Computing Applications*, Vol. 21, No. 3, 2007, pp. 266-290.

[68] Laxmikant V. Kale, Sanjeev Krishnan, "CHARM++: a portable concurrent object oriented system based on C++," *ACM SIGPLAN Notices*, Vol. 28, No. 10, 1993, pp. 91-108.

[69] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally,

Pat Hanrahan,"Sequoia: Programming the Memory Hierarchy," *ACM/IEEE SC 2006 Conference* (SC'06), 2006, Article No. 83.

[70] Charles Koelbel, David B. Loveman, Guy L. Steele, Mary E. Zosel, *High Performance FORTRAN Handbook*, MIT Press, 1994, pp. 1-329.

[71] Paras Mehta, José Nelson Amaral, Duane Szafron, "Is MPI Suitable for a Generative Design-Pattern System?" *Parallel Computing*, Vol. 32, Nos. 7-8, 2006, pp. 616-626.

[72] Alberto Bartoli, Paolo Corsini, Gianluca Dini, Cosimo Antonio Prete, "Graphical Design of Distributed Applications through Reusable Components," *IEEE Parallel and Distributed Techonology*, Vol. 3, No. 1, 1995, pp. 37-51.

[73] Bryan Catanzaro, Armando Fox, Kurt Keutzer, David Patterson, Bor-Yiing Su, Marc Snir, Kunle Olukotun, Pat Hanrahan, Hassan Chafi, "Ubiquitous Parallel Computing from Berkeley, Illinois, and Stanford," *IEEE Micro*, Vol. 30, No. 2, 2010, pp. 41-55.

[74] Jeffery Dean, Sanjay Ghemawat, "MapReduce: Simplifed Data Processing on Large Clusters," *Communications of the ACM* , Vol. 51, No. 1, 2008, pp. 107-113.

[75] Hadoop MapReduce: http://hadoop.apache.org/common/docs/current/

[76] Blackford, L. S., Choi J., Cleary A., D'Azevedo E., Jemmel J., Dhillon I., Dongarra J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., and Whaley, R.C., "ScaLAPACK Users' Guide," *SIAM*, 1997, pp.1-325.

[77] POOMA User Guide: http://acts.nersc.gov/pooma/

[78] PETSc Webpage: *http://www.mcs.anl.gov/petsc/petsc-as/*

[79] Mark. T. Jones, Paul E. Plassmann, "BlockSolve95 users manual: Scalable library software for the parallel solution of sparse linear systems," *ANL Report ANL-95/48, Argonne National Laboratory*, 1995, pp. 1-41.

[80] Roger D. Chamberlain, Mark A. Franklin, Eric J. Tyson, Jeremy Buhler, Saurabh Gayen, Patrick Crowley, James H. Buckley, "Application development on hybrid systems," SC 2007.

[81] Jeff Gray, Suman Roychoudhury, "A Technique for Constructing Aspect Weavers using a Program Transformation Engine," *International Conference on Aspect-Oriented Software Development*, 2004, pp. 36-45.

[82] Balakrishna Ramkumar, Volker Strumpen, "Portable Checkpointing for Heterogeneous Architectures," *27th International Symposium on Fault-Tolerant Computing - Digest of Papers*, 1997, pp. 58-67.

[83] Hai Jiang, Vipin Chaudhary, "MigThread: Compile/runtime support for thread migration," *Proceedings of International Parallel and Distributed Processing Symposium*, IPDPS 2002, pp. 58-66.

[84] Pawel Czarnul, Marcin Fraczak, "New User-Guided and ckpt-Based Checkpointing Libraries for Parallel MPI Applications," *12th European PVM/MPI Users' Group Meeting*, Vol. LNCS 3666, 2005, pp. 351- 358.

[85] James Cordy, Thomas Dean, Andrew Malton, and Kevin Schneider, "Source Transformation in Software Engineering using the TXL Transformation System," *Journal of Information and Software Technology*, Vol. 44, No. 13, 2002, pp. 827-837.

[86] Eelco Visser, "Stratego: A Language for Program Transformation Based on Rewriting Strategies. System Description of Stratego 0.5," *International Conference on Rewriting Techniques and Applications,* Springer-Verlag LNCS 2051, Utrecht, Netherlands, May 2001, pp. 357-361.

[87] Mark van den Brand, Jan Heering, Paul Klint, and Pieter Olivier, "Compiling Rewrite Systems: The ASF+SDF Compiler," *ACM Transactions on Programming Languages and Systems*, Vol. 24, No. 4, 2002, pp. 334-368.

[88] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill, *A Pattern Language for Parallel Programming*, Addison Wesley Software Patterns Series, 2004.

[89] Michael Quinn, *Parallel programming in C with MPI and OpenMP*, McGraw-Hill, 2004.

[90] Chung, T. J, *Computational Fluid Dynamics*, Cambridge University Press, 1st edition, 2002.

[91] Barry Wilkinson, Michael Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations*, Prentice Hall, 1998, pp. 1-431.

[92] Chengcui Zhang, Xin Chen, "Region Based Image Clustering and Retrieval using Multiple Instance Learning," *Lecture Notes in Computer Science, Image/Video Annotation and Clustering*, 2005, pp. 194-204.

[93] Ritu Arora, and Purushotham Bangalore, "Grid enabling a Content Based Image Retrieval Application," *International Conference on Parallel and Distributed Computing Systems (ISCA PDCS 2007),* 2007, pp. 19-23.

[94] Ferosh Jacob, Ritu Arora, Purushotham Bangalore, Marjan Mernik, Jeff Gray, "Raising the level of abstraction of GPU-programming," *Parallel and Distributed Processing Techniques and Applications (PDPTA 2010)*, pp. 339-345.

[95] Intel Vtune: http://software.intel.com/en-us/intel-vtune/

[96] Craig C. Douglas, Gundolf Haase, Jonathan Hu, Markus Kowarschik, Ulrich Rüde, Christian Weiß, "Portable Memory Hierarhy Tehniques for PDE Solvers," *SIAM News*, Vol. 33, 2000, pp. 8-9.

[97] Guy E. Blelloch, Rezaul A. Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch, "Provably Good Multicore Cache Performance for Divide-and-Conquer Algorithms," In *Proceedings of the 19th ACM-SIAM Symposium on Discrete Algorithms (SODA'08)*, 2008.

APPENDIX A

HI-PAL METAMODEL SPECIFICATIONS

The individual specifications within this Appendix show the KM3 and TCS specifications for the Hi-PaL metamodel.

A.1. Hi-PaL Metamodel KM3 Specification

The following represents the excerpt of the KM3 specification for the Hi-PaL metamodel.

```
-- @name           PDSL
-- @version        1.0

package PDSL {

      -- extend LocatedElement class.

-- BEGIN DSL-specific classes
    class PDSL extends LocatedElement {
            reference parSpecs[*] container : ParSpecs;
            reference hookType container : HookType;
            attribute pattern : String;
            attribute mapping : String;
      }
      class ParSpecs extends LocatedElement {
            reference parTask [*] container : ParTask;
            reference parCond[*] container : ParCond;
      }
      class ParCond extends LocatedElement {
          reference hook container : Hook;
            attribute pattern : String;
      }
      class Hook extends LocatedElement {
            reference hookType container : HookType;
            reference hookElem container : HookElement;
      }

      abstract class HookType extends LocatedElement {
      }


      abstract class HookElement extends LocatedElement {
      }

      class Statement extends HookElement {

      }
      class Call extends HookElement {

      }
```

```
class Execution extends HookElement {

}
class Function extends HookElement {

}
abstract class ParTask extends LocatedElement {
}

abstract class ParCompute extends ParTask {
}

abstract class ParComputeLimits extends ParCompute {
}

class ParReduce extends ParTask {
    reference redVarType container : RedVarType;
      reference varArgs[*] container : RedVarArg;
}


class ParAllReduce extends ParTask {
    reference redVarType container : AllRedVarType;
    reference varArgs[*] container : RedVarArg;
}


class ParFor extends ParTask {
   reference forLoopInitStatement container : ForInitStatement;
   reference forLoopCond container : ForCond;
   reference forLoopExpr container : ForLoopExpression;
}
abstract class ForInitStatement extends LocatedElement {
}
class InitStatement extends ForInitStatement {
      attribute forVar : String;
      reference operator container: Operator;
      reference limit container : ParComputeLimits;
}
class AnyStatement extends ForInitStatement {
      attribute anyStatement : String;
}
abstract class Operator extends LocatedElement {

}

abstract class ForCond extends LocatedElement {
}
class ForCondPresent extends ForCond{
      attribute forVar : String;
      reference operator container: Operator;
      reference limit container : ParComputeLimits;
}
class ForNoCond extends ForCond {
}
class AnyCondition extends ForCond {
      attribute anyCondition : String;
```

```
	}

	class LoopExpression extends ForLoopExpression {
		attribute forVar : String;
		reference stride container : Stride;
	}

	class ParGather extends ParTask {
	    reference gatherVarType container : ParGatherArrayType;
		reference varArgs[*] container : GatherVarArg;
	}
	class ParDistribute extends ParTask {
	    reference distributeVarType container :
	                                    ParDistributeArrayType;
	    reference varArgs[*] container : DistributeVarArg;
	}

	class ParExchange extends ParTask {
		reference exchangeVarType container : ParExchangeArrayType;
		reference varArgs[*] container : ExchangeVarArg;
	}


	class ParBroadCast extends ParTask {
	    reference broadcastVarType container : ParBroadCastArrayType;
	    reference varArgs[*] container : BroadCastVarArg;
	}

	class ParWrite extends ParTask {
		reference wVarType container : WriteVarType;
		reference varArgs[*] container : WriteVarArg;
	}

	class ParRead extends ParTask {
		reference rVarType container : ReadVarType;
		reference varArgs[*] container : ReadVarArg;
	}



	-- More DSL-specific classes


-- END DSL-specific classes
}
```

A.2. Hi-PaL TCS Specification

The following shows the excerpt of the TCS specification for the Hi-PaL metamodel. The lexical part is not included here.

```
syntax PDSL {

-- BEGIN Primitive templates
-- Specifies representation of primitive types.
-- Only needs modification when default lexer is not satisfactory.
-- Generally modified along with the lexer.
      primitiveTemplate identifier for String default using NAME:
            value = "%token%";

      primitiveTemplate stringSymbol for String using STRING:
            value = "%token%",
            serializer="'\'' + %value%.toCString() + '\''";

      primitiveTemplate integerSymbol for Integer default using INT:
            value = "Integer.valueOf(%token%)";

      primitiveTemplate floatSymbol for Double default using FLOAT:
            value = "Double.valueOf(%token%)";
-- END Primitive templates

-- BEGIN Class templates
-- Specifies representation of classes.
-- This is the main section to work on.


      template PDSL main
            :     "Parallel" "section" "begins" hookType "(" pattern
                  ")"  "mapping" "is" mapping "{"
                           parSpecs {separator = ";"}
                        "}"
            ;
      template ParSpecs
                  :     parTask parCond {separator = "&&"}
            ;
      template ParCond
                  :     hook "(" pattern ")"
            ;

      template Hook
                  : hookType hookElem
            ;
      template HookType abstract;

      template HookElement abstract;
      template Statement
            :
            "statement"
          ;
      template Call
            :
            "call"
          ;
      template Execution
            :
            "execution"
          ;
      template Function
```

```
      :
      "function"
    ;
template ParTask abstract;

template ParCompute abstract;

template ParComputeLimits abstract;

template ParReduce
      : redVarType "(" varArgs{separator = ","} ")"
    ;

template ParAllReduce
      : redVarType "(" varArgs{separator = ","} ")"
    ;

template ParFor
      : "Parallelize_For_Loop"  "where" "(" forLoopInitStatement
        ";" forLoopCond ";" forLoopExpr ")"
    ;

template ForInitStatement abstract;

template InitStatement
      : forVar operator limit
    ;

template AnyStatement
      : anyStatement
    ;

template Operator abstract;

template ForCond abstract;

template ForCondPresent
      : forVar operator limit
    ;

template ForNoCond
      : ";"
    ;

template AnyCondition
      : anyCondition
    ;

template ForLoopExpression abstract;

template ParGather
      : gatherVarType "(" varArgs{separator = ","} ")"
    ;

template ParDistribute
      : distributeVarType "(" varArgs{separator = ","} ")"
    ;
```

```
        template ParExchange
              : exchangeVarType "(" varArgs{separator = ","} ")"
            ;

        template ParBroadCast
              : broadcastVarType "(" varArgs{separator = ","} ")"
            ;

        template ParWrite
              : wVarType "(" varArgs{separator = ","} ")"
            ;

        template ParRead
              : rVarType "(" varArgs{separator = ","} ")"
            ;


-- More Class templates corresponding to the KM3 elements not shown in
-- A.2.

-- END Class templates
```

APPENDIX B

DALC METAMODEL SPECIFICATIONS

The individual specifications within this Appendix show the KM3 and TCS specifications for DALC.

B.1. DALC KM3 Specification

The following represents the excerpt of the KM3 specification for the DALC metamodel.

```
package CDSL {

      -- LocatedElement class

-- BEGIN DSL-specific classes

      class CDSL extends LocatedElement {
            reference checkptCond container : ChkCond;
            reference checkptCode container : ChkCode;
            reference restartCond container : RestartCond;
            reference restartCode container : RestartCode;
      }
      class RestartCond extends LocatedElement {
            reference hook container : Hook;
            attribute pattern : String;
      }
      class RestartCode extends LocatedElement {
            reference restartStmts[*] container : RestartStmt;
      }

      abstract class RestartStmt extends LocatedElement {
            reference rVarType container : ReadVarType;
            reference varArgs[*] container : RestartVarArg;
      }

      class RestartStmt1 extends RestartStmt {
      }


      class RestartStmt2 extends RestartStmt {
            reference rVarType1 container : ReadVarType;
            reference varArgs1[*] container : RestartVarArg;
      }


     class RestartStmt3 extends RestartStmt {
           attribute name : String;
           attribute _params[*] : String;
      }
```

```
      class RestartVarArg extends LocatedElement {
            attribute argument : String;
       }

       class ChkCond extends LocatedElement {
           reference hook container : Hook;
            attribute pattern : String;
            attribute frequency : Integer;
            attribute loopVar : String;
       }

       class Hook extends LocatedElement {
            reference hookType container : HookType;
            reference hookElem container : HookElement;
       }

       class ChkCode extends LocatedElement {
            reference checkptStmts[*] container : ChkStmt;
       }

       class ChkStmt extends LocatedElement {
            reference sVarType container : SaveVarType;
            reference varArgs[*] container : SaveVarArg;
       }

       class SaveVarArg extends LocatedElement {
            attribute argument : String;
       }

       abstract class HookType extends LocatedElement {
       }

       abstract class HookElement extends LocatedElement {
       }

       class Statement extends HookElement {

       }
       class Call extends HookElement {

       }
       class Execution extends HookElement {

       }

       abstract class SaveVarType extends LocatedElement {
       }


       abstract class ReadVarType extends LocatedElement {
       }

       -- More DSL-specific classes
-- END DSL-specific classes

}
```

B.2. DALC TCS Specification

The following shows the excerpt of the TCS specification for the DALC metamodel. The lexical part is not included here.

```
syntax CDSL {

-- BEGIN Primitive templates
-- Specifies representation of primitive types.
-- Only needs modification when default lexer is not satisfactory.
-- Generally modified along with the lexer.
      primitiveTemplate identifier for String default using NAME:
            value = "%token%";

      primitiveTemplate stringSymbol for String using STRING:
            value = "%token%",
            serializer="'\'' + %value%.toCString() + '\''";

      primitiveTemplate integerSymbol for Integer default using INT:
            value = "Integer.valueOf(%token%)";

      primitiveTemplate floatSymbol for Double default using FLOAT:
            value = "Double.valueOf(%token%)";
-- END Primitive templates

-- BEGIN Class templates
-- Specifies representation of classes.
-- This is the main section to work on.


      template CDSL main
            :     "beginCheckpointing" ":"
                        checkptCond "{"
                        checkptCode
                  "}"
                  "beginInitialization" ":"
                        restartCond  "{"
                        restartCode
                  "}"
            ;
      template RestartCond
                  :     hook "(" pattern ")"
            ;

      template ChkCond
                  :     hook "(" pattern ")" "&&"
                     "(" "frequency" "=" frequency ")"
                     "&&" "(" "loopVar" "=" loopVar ")"
```

```
      ;

template Hook
            : hookType hookElem
      ;

template ChkCode
      : checkptStmts
    ;
template RestartCode
      : restartStmts
    ;
template ChkStmt
      : sVarType "(" varArgs{separator = ","} ")"
    ;
template RestartStmt abstract;


template RestartStmt3
      : rVarType "(" varArgs{separator = ","} ")" "|"
        name "<" _params {separator = ","} ">" "("
        varArgs{separator = ","} ")"
    ;

template RestartStmt1
      : rVarType "(" varArgs{separator = ","} ")"
    ;

template RestartStmt2
      : rVarType "(" varArgs{separator = ","} ")" "|"
        rVarType1 "(" varArgs1{separator = ","} ")"
    ;

template RestartVarArg
      : argument
    ;
template SaveVarArg
      : argument
    ;

template HookType abstract;

template HookElement abstract;

template SaveVarType abstract;

template ReadVarType abstract;


template Statement
      :
      "statement"
    ;
template Call
      :
      "call"
    ;
```

```
      template Execution
            :
            "execution"
          ;



    -- More class templates corresponding to the KM3 elements not shown
in B.1.

-- END Class templates
```

APPENDIX C

MODEL TRANSFORMATION RULES FOR HI-PAL AND DALC

The individual specifications within this Appendix show a sample of the model transformation rules for the Hi-PaL and DALC.

C.1. ATL Rule for Setting the MPI Environment in Hi-PaL

The following ATL rule shows the complete specification for generating the RSL rules for inserting the MPI-library calls at specific points in the existing sequential code.

```
module PSDL2RSL;

create OUT : RSL3 from IN : PDSL;

rule PSDL2RSL {
      from
            s : PDSL!PDSL
      to
            t : RSL3!RSL3 (
                  domain <- dom,
                  rslelems <- Sequence {pat1, expat1, rule1, pat2,
                              pat3, expat2, rule2, pat4, pat5, expat3,
                              rule3, pat6, expat4, rule4},
                  ruleset <- rs
            ),
            dom : RSL3!Domain  (
                  dname <- 'Cpp'
            ),
            rs : RSL3!RuleSet (
                  rsname <- 'r',
                  rname <- Sequence {'addIncludeFile', 'extend_decl',
                           'add_statements', 'change_exit'}
            ),
            pat1 : RSL3!Pattern(
                  phead <- ph,
                  ptoken <- 'statement_seq',
                  ptext <- pt
                  ),
            ph : RSL3!PatternHead (
                  name <- 'add_var'
            ),
            pt : RSL3!SimplePatternText (
                  ptext <-   ' \\>Cpp\\:[simple_declaration =
                              decl_specifier_seq init_declarator_list
                              \';\'] int \\>Cpp\\:[declarator_id =
                              id_expression] rank_Fraspa
                              \\<\\:declarator_id ;
                              \\<\\:simple_declaration
                              \\>Cpp\\:[simple_declaration =
                              decl_specifier_seq init_declarator_list
```

```
                \';\'] int \\>Cpp\\:[declarator_id =
                id_expression] size_Fraspa
                \\<\\:declarator_id ;
                \\<\\:simple_declaration
                  '

),
expat1 : RSL3!ExternalPattern(
      dname <-'Cpp',
      eptext <- 'addVars' ,
      phead <- ph1,
      ptoken <- 'translation_unit'
),
ph1 : RSL3!PatternHead(
      name <- 'addVars',
      params <- Sequence{param1, param2}
),
param1 : RSL3!PatternParameter(
       name <- 'tu' ,
       referTo <- 'translation_unit'

),
param2 : RSL3!PatternParameter(
       name <- 'stmt_seq' ,
       referTo <- 'statement_seq'

),
rule1 :RSL3!Rule (
   rname <- 'extend_decl',
   params <- Sequence{rlparam1},
   type <- 'translation_unit',
   r_lhs_pattern  <- lhs1,
   r_rhs_pattern <- rhs1
),
rlparam1 : RSL3!PatternParameter(
       name <- 'tu' ,
       referTo <- 'translation_unit'

),
lhs1 : RSL3!RuleLHS(
       ruletext <- text1
),
rhs1 : RSL3!RuleRHS(
       ruletext  <- text2,
       condition <- Sequence {rulecond1}
),
text1 : RSL3!IDRuleText(
      text <-'tu'
),
text2 : RSL3!ComplexRuleText(
      pref <- pr1
),
pr1 : RSL3!PatternRef (
      name <- 'addVars',
      params <- Sequence{param01, param02}
),
param01 : RSL3!RealParameter(
```

```
        name <- 'tu'

),
param02 : RSL3!PatternRef(
        name <- 'add_var'
),
rulecond1 : RSL3!RuleNotEqCondition(
      lhs <- 'tu',
      pref <- pr
),
pr : RSL3!PatternRef (
      name <- 'addVars',
      params <- Sequence{param11, param21}
),
param11 : RSL3!RealParameter(
        name <- 'tu'

),
param21 : RSL3!PatternRef(
        name <- 'add_var'
),

pat2 : RSL3!Pattern(
      phead <- ph2,
      ptoken <- 'statement_seq',
      ptext <- pt2
      ),
ph2 : RSL3!PatternHead (
      name <- 'add_code1'
),
pt2 : RSL3!SimplePatternText (
      ptext <-   '
                  \\>Cpp\\:[ postfix_expression =
                  postfix_expression \'(\' expression_list
                  \')\'] MPI_Init(NULL,NULL)
                  \\<\\:postfix_expression ;
                  \\>Cpp\\:[ postfix_expression =
                  postfix_expression \'(\' expression_list
                  \')\'] MPI_Comm_size( MPI_COMM_WORLD,
                  &size_Fraspa ) \\<\\:postfix_expression ;
                  \\>Cpp\\:[ postfix_expression =
                  postfix_expression \'(\' expression_list
                  \')\'] MPI_Comm_rank( MPI_COMM_WORLD,
                  &rank_Fraspa ) \\<\\:postfix_expression ;
                '

),
pat3 : RSL3!Pattern(
      phead <- ph3,
      ptoken <- 'statement',
      ptext <- pt3
      ),
ph3 : RSL3!PatternHead (
      name <- 'search_pattern1'
),
pt3 : RSL3!SimplePatternText (
      ptext <-    if
```

```
                (s.pattern.substring(s.pattern.indexOf('(') -
                > abs(), s.pattern.indexOf(')')->abs() ) -
                > size() < 2) then
                      '\\>Cpp\\:[ expression =
                       assignment_expression] '
              s.pattern.substring(1,(s.pattern.indexOf('='))) +

             ' = ' +
             s.pattern.substring(((s.pattern.indexOf('='))+2),
            (s.pattern -> size() -1 ))+' \\<\\:expression ;
                                        '
           else if (s.pattern.substring(
                    s.pattern.indexOf('('),
                    s.pattern.indexOf(')') )->size()  = 2)
                then
                 '\\>Cpp\\:[ expression =
                 assignment_expression] '+
                s.pattern.substring(1,(s.pattern.indexOf('=')))
                +' = \\>Cpp\\:[ postfix_expression =
                simple_type_specifier \'(\' \')\']  '+
                s.pattern.substring(((s.pattern.indexOf('='))
                +2), (s.pattern -> size() -1 ))+'
                \\<\\:postfix_expression \\<\\:expression ;
                '
                else if (s.pattern.substring(
                        s.pattern.indexOf('('),
                        s.pattern.indexOf(')') )->size() > 2)
                     then
                      '\\>Cpp\\:[ expression =
                       assignment_expression] '+
               s.pattern.substring(1,(s.pattern.indexOf('=')))
               +' = \\>Cpp\\:[ postfix_expression =
               postfix_expression \'(\' expression_list \')\']
               '+
               s.pattern.substring(((s.pattern.indexOf('='))+2),
               (s.pattern -> size() -1 ))+'
               \\<\\:postfix_expression \\<\\:expression ;
                                        '
                    else
                      ''
                    endif
                endif
        endif

),
expat2 : RSL3!ExternalPattern(
      dname <-'Cpp',
      eptext <- if
              (s.hookType.oclIsTypeOf(PDSL!BeforeHookType))
                then
                    'addCodeBeforeStatement'
                else
                    'addCodeAfterStatement'
                endif,

      phead <- ph4,
      ptoken <- 'translation_unit'
```

```
),
ph4 : RSL3!PatternHead(
      name <- if
              (s.hookType.oclIsTypeOf(PDSL!BeforeHookType))
              then
                  'addCodeBeforeStatement'
              else
                  'addCodeAfterStatement'
              endif,
      params <- Sequence{param41, param42,param43, param44,
                param45}
),
param41 : RSL3!PatternParameter(
       name <- 'tu' ,
       referTo <- 'translation_unit'

),
param42 : RSL3!PatternParameter(
       name <- 'stmt' ,
       referTo <- 'statement'

),
param43 : RSL3!PatternParameter(
       name <- 's_seq2' ,
       referTo <- 'statement_seq'

),
param44 : RSL3!PatternParameter(
       name <- 'id' ,
       referTo <- 'IDENTIFIER'

),
param45 : RSL3!PatternParameter(
       name <- 'id2' ,
       referTo <- 'IDENTIFIER'

),
rule2 :RSL3!Rule (
   rname <- 'add_statements',
   params <- Sequence{r2param1},
   type <- 'translation_unit',
   r_lhs_pattern  <- lhs2,
   r_rhs_pattern <- rhs2
),
r2param1 : RSL3!PatternParameter(
       name <- 'tu' ,
       referTo <- 'translation_unit'

),
lhs2 : RSL3!RuleLHS(
       ruletext <- text21
),
text21 : RSL3!IDRuleText(
      text <-'tu'
),
rhs2 : RSL3!RuleRHS(
       ruletext  <- text22,
```

```
       condition <- Sequence {rulecond2}
),
text22 : RSL3!ComplexRuleText(
      pref <- pr2
),
pr2 : RSL3!PatternRef (
      name <- if
             (s.hookType.oclIsTypeOf(PDSL!BeforeHookType))
             then
                  'addCodeBeforeStatement'
             else
                  'addCodeAfterStatement'
             endif,
      params <- Sequence{param201, param202, param203,
                param204, param205}
),
param201 : RSL3!RealParameter(
       name <- 'tu'

),
param202 : RSL3!PatternRef(
       name <- 'search_pattern1'
),
param203 : RSL3!PatternRef(
       name <- 'add_code1'
),
param204 : RSL3!StringParameter(
       name <- '' +  'main' + ' '
),
param205 : RSL3!StringParameter(
       name <- ''+
      s.pattern.substring(1,(s.pattern.indexOf('='))) + ' '
),
rulecond2 : RSL3!RuleNotEqCondition(
      lhs <- 'tu',
      pref <- pr3
),
pr3 : RSL3!PatternRef (
      name <- if
         (s.hookType.oclIsTypeOf(PDSL!BeforeHookType)) then
                 'addCodeBeforeStatement'
              else
                  'addCodeAfterStatement'
              endif,
      params <- Sequence{param211, param212,param213,
                param214, param215 }
),
param211 : RSL3!RealParameter(
       name <- 'tu'

),
param212 : RSL3!PatternRef(
       name <- 'search_pattern1'
),
param213 : RSL3!PatternRef(
       name <- 'add_code1'
),
```

```
param214 : RSL3!PatternRef2(
       ptext <- param2214
),
param215 : RSL3!PatternRef2(
       ptext <- param2215
),
param2214 : RSL3!StringParameter(
       name <- '' +  'main' + ' '
),

param2215 : RSL3!StringParameter(
      name <- ''+
      s.pattern.substring(1,(s.pattern.indexOf('='))) + ' '
),
pat4 : RSL3!Pattern(
      phead <- ph5,
      ptoken <- 'jump_statement',
      ptext <- pt4
      ),
ph5 : RSL3!PatternHead (
      name <- 'returnStmt'
),
pt4 : RSL3!SimplePatternText (
      ptext <-   'return 0;'

),
pat5 : RSL3!Pattern(
      phead <- ph6,
      ptoken <- 'statement',
      ptext <- pt5
      ),
ph6 : RSL3!PatternHead (
      name <- 'add_finalize_stmt'
),
pt5 : RSL3!SimplePatternText (
      ptext <-   '\\>Cpp\\:[ postfix_expression =
                 postfix_expression \'(\'  \')\']
                 MPI_Finalize() \\<\\:postfix_expression ;'
),
expat3 : RSL3!ExternalPattern(
      dname <-'Cpp',
      eptext <- 'addFinalize' ,
      phead <- ph7,
      ptoken <- 'translation_unit'
),
ph7 : RSL3!PatternHead(
      name <- 'addFinalize',
      params <- Sequence{param71, param72,param73}
),
param71 : RSL3!PatternParameter(
       name <- 'tu' ,
       referTo <- 'translation_unit'

),
param72 : RSL3!PatternParameter(
       name <- 'jstmt' ,
       referTo <- 'jump_statement'
```

```
),
param73 : RSL3!PatternParameter(
       name <- 'stmt2' ,
       referTo <- 'statement'

),
rule3 :RSL3!Rule (
   rname <- 'change_exit',
   params <- Sequence{r3param1},
   type <- 'translation_unit',
   r_lhs_pattern  <- lhs3,
   r_rhs_pattern <- rhs3
),
r3param1 : RSL3!PatternParameter(
       name <- 'tran_unit' ,
       referTo <- 'translation_unit'

),
lhs3 : RSL3!RuleLHS(
       ruletext <- text31
),
text31 : RSL3!IDRuleText(
      text <-'tran_unit'
),
rhs3 : RSL3!RuleRHS(
       ruletext  <- text32,
       condition <- Sequence {rulecond3}
),
text32 : RSL3!ComplexRuleText(
      pref <- pr4
),
pr4 : RSL3!PatternRef (
      name <- 'addFinalize',
      params <- Sequence{param301, param302, param303}
),
param301 : RSL3!RealParameter(
       name <- 'tran_unit'

),
param302 : RSL3!PatternRef(
       name <- 'returnStmt'
),
param303 : RSL3!PatternRef(
       name <- 'add_finalize_stmt'
),
rulecond3 : RSL3!RuleNotEqCondition(
      lhs <- 'tran_unit',
      pref <- pr5
),
pr5 : RSL3!PatternRef (
      name <- 'addFinalize',
      params <- Sequence{param311, param312,param313}
),
param311 : RSL3!RealParameter(
       name <- 'tran_unit'
```

```
),
param312 : RSL3!PatternRef(
       name <- 'returnStmt'
),
param313 : RSL3!PatternRef(
       name <- 'add_finalize_stmt'
),
pat6 : RSL3!Pattern(
      phead <- ph8,
      ptoken <- 'pp_declaration_seq',
      ptext <- pt6
      ),
ph8 : RSL3!PatternHead (
      name <- 'fileToInclude'
),
pt6 : RSL3!SimplePatternText (
      ptext <-   '\\>Cpp\\:[pp_declaration_seq =
                  control_line]
                        #include <mpi.h> \\&n

                  \\<\\:pp_declaration_seq'
),
expat4 : RSL3!ExternalPattern(
      dname <-'Cpp',
      eptext <- 'IncludeFile2' ,
      phead <- ph9,
      ptoken <- 'translation_unit'
),
ph9 : RSL3!PatternHead(
      name <- 'IncludeFile2',
      params <- Sequence{param91, param92}
),
param91 : RSL3!PatternParameter(
       name <- 'tran_unit' ,
       referTo <- 'translation_unit'

),
param92 : RSL3!PatternParameter(
       name <- 'pep_dec_seq' ,
       referTo <- 'pp_declaration_seq'

),
rule4 :RSL3!Rule (
   rname <- 'addIncludeFile',
   params <- Sequence{r4param1},
   type <- 'translation_unit',
   r_lhs_pattern  <- lhs4,
   r_rhs_pattern <- rhs4
),
r4param1 : RSL3!PatternParameter(
       name <- 'tran_unit' ,
       referTo <- 'translation_unit'

),
lhs4 : RSL3!RuleLHS(
       ruletext <- text41
),
```

```
                text41 : RSL3!IDRuleText(
                      text <-'tran_unit'
                ),
                rhs4 : RSL3!RuleRHS(
                       ruletext  <- text42,
                       condition <- Sequence {rulecond4}
                ),
                text42 : RSL3!ComplexRuleText(
                      pref <- pr6
                ),
                pr6 : RSL3!PatternRef (
                      name <- 'IncludeFile2',
                      params <- Sequence{param401, param402}
                ),
                param401 : RSL3!RealParameter(
                       name <- 'tran_unit'

                ),
                param402 : RSL3!PatternRef(
                       name <- 'fileToInclude'
                ),
                rulecond4 : RSL3!RuleNotEqCondition(
                      lhs <- 'tran_unit',
                      pref <- pr7
                ),
                pr7 : RSL3!PatternRef (
                      name <- 'IncludeFile2',
                      params <- Sequence{param411, param412}
                ),
                param411 : RSL3!RealParameter(
                       name <- 'tran_unit'

                ),
                param412 : RSL3!PatternRef(
                       name <- 'fileToInclude'
                )

}
```

C.2. ATL Rule for Translating DALC code into RSL code

The following ATL rule shows the specification for translating one of the DALC specifications for checkpointing and restart into low-level RSL code.

```
modulemodule CSDL2RSL;

create OUT : RSL from IN : CDSL;

rule poisson {
      from
            s : CDSL!CDSL
```

```
to
      t : RSL!RSL (
            domain <- dom,
            rslelems <- Sequence {pat1, pat2, pat3, pat4,
                        expat,expat2,rule1,rule2},
            ruleset <- rs
      ),
      dom : RSL!Domain  (
            dname <- 'Cpp~VisualCpp6'
      ),
      rs : RSL!RuleSet (
            rsname <- 'r',
            rname <- Sequence
                     {'change_statement','change_statement2'}
      ),
      pat1 : RSL!Pattern(
            phead <- ph,
            ptoken <- 'statement_seq',
            ptext <- pt
            ),
      ph : RSL!PatternHead (
            name <- 'chk_code'
      ),

      pat2 : RSL!Pattern(
            phead <- ph2,
            ptoken <- 'statement_seq',
            ptext <- pt2
            ),
      ph2 : RSL!PatternHead (
            name <- 'add_code'
      ),


      expat : RSL!ExternalPattern(
            dname <-'Cpp~VisualCpp6',
            eptext <- 'modify' ,
            phead <- ph3,
            ptoken <- 'translation_unit'
            --ptext <- pt3
      ),
      ph3 : RSL!PatternHead(
            name <- 'modify',
            params <- Sequence{param1, param2, param3, param4}
      ),
      param1 : RSL!PatternParameter(
             name <- 'tu' ,
             referTo <- 'translation_unit'

      ),
      param2 : RSL!PatternParameter(
             name <- 'stmt_seq' ,
             referTo <- 'statement_seq'

      ),
      param3 : RSL!PatternParameter(
             name <- 's_seq2' ,
```

```
        referTo <- 'statement_seq'

),
param4 : RSL!PatternParameter(
        name <- 'id' ,
        referTo <- 'IDENTIFIER'

),
expat2 : RSL!ExternalPattern(
      dname <-'Cpp~VisualCpp6',
      eptext <- 'delModify' ,
      phead <- ph33,
      ptoken <- 'translation_unit'
),
ph33 : RSL!PatternHead(
      name <- 'delModify',
      params <- Sequence{param13, param23, param33,
                 param43, param53}
),
param13 : RSL!PatternParameter(
        name <- 'tu' ,
        referTo <- 'translation_unit'

),
param23 : RSL!PatternParameter(
        name <- 's_seq' ,
        referTo <- 'statement_seq'

),
param33 : RSL!PatternParameter(
        name <- 's_seq2' ,
        referTo <- 'statement_seq'

),
param43 : RSL!PatternParameter(
        name <- 'id1' ,
        referTo <- 'IDENTIFIER'

),
param53 : RSL!PatternParameter(
        name <- 'id2' ,
        referTo <- 'IDENTIFIER'

),
rule1 :RSL!Rule (
   rname <- 'change_statement',
   params <- Sequence{rlparam1},
   type <- 'translation_unit',
   r_lhs_pattern  <- lhs1,
   r_rhs_pattern <- rhs1
),
rlparam1 : RSL!PatternParameter(
        name <- 'tu' ,
        referTo <- 'translation_unit'

),
lhs1 : RSL!RuleLHS(
```

```
       ruletext <- text1
),
rhs1 : RSL!RuleRHS(
       ruletext  <- text2,
       condition <- Sequence {rulecond1}
),
text1 : RSL!IDRuleText(
      text <-'tu'
),
text2 : RSL!ComplexRuleText(
      pref <- pr1
),
pr1 : RSL!PatternRef (
      name <- 'delModify',
      params <- Sequence{param01, param02, param03,
                param04,param05}
),
param01 : RSL!RealParameter(
       name <- 'tu'

),
param02 : RSL!PatternRef(
       name <- 'myStart'
),
param03 : RSL!PatternRef(
       name <- 'add_code'

),
param04 : RSL!StringParameter(
       name <- '' + s.restartCond.pattern.substring(1,5) +
               ' '

),
param05 : RSL!StringParameter(
       name <- '' + s.restartCode.restartStmts->at(1).name
              + ' '

),
rulecond1 : RSL!RuleNotEqCondition(
      lhs <- 'tu',
      pref <- pr
),
pr : RSL!PatternRef (
      name <- 'delModify',
      params <- Sequence{param11, param21, param31,
                param41, param51}
),
param11 : RSL!RealParameter(
       name <- 'tu'

),
param21 : RSL!PatternRef(
       name <- 'myStart'
),
param31 : RSL!PatternRef(
       name <- 'add_code'
```

```
),
param41 : RSL!StringParameter(
       name <-  '' + s.restartCond.pattern.substring(1,5) +
                ' '

),
param51 : RSL!StringParameter(
        name <- '' + s.restartCode.restartStmts->at(1).name
              + ' '

),
pat3 : RSL!Pattern(
      phead <- ph4,
      ptoken <- 'statement_seq',
      ptext <- pt4
      ),
ph4 : RSL!PatternHead (
      name <- 'search_pattern'
),
pat4 : RSL!Pattern(
      phead <- pah4,
      ptoken <- 'statement_seq',
      ptext <- pt5
      ),
pah4 : RSL!PatternHead (
      name <- 'myStart'
),

rule2 : RSL!Rule (
   rname <- 'change_statement2',
   params <- Sequence{rlparam2},
   type <- 'translation_unit',
   r_lhs_pattern  <- lhs2,
   r_rhs_pattern <- rhs2
),
rlparam2 : RSL!PatternParameter(
       name <- 'tu' ,
       referTo <- 'translation_unit'
),
lhs2 : RSL!RuleLHS(
      ruletext <- text3
),
text3 : RSL!IDRuleText(
      text <-'tu'
),
rhs2 : RSL!RuleRHS(
      ruletext  <- text4,
      condition <- Sequence {rulecond2}
),
text4 : RSL!ComplexRuleText(
      pref <- pr11
),
pr11 : RSL!PatternRef (
      name <- 'modify',
      params <- Sequence{param011, param021, param031,
                param041}
),
```

```
                param011 : RSL!RealParameter(
                       name <- 'tu'

                ),
                param021 : RSL!PatternRef(
                       name <- 'search_pattern'
                ),
                param031 : RSL!PatternRef(
                       name <- 'chk_code'

                ),
                param041 : RSL!StringParameter(
                       name <- '' + s.checkptCond.pattern.substring(1,4) +
                              ' '

                ),
                rulecond2 : RSL!RuleNotEqCondition(
                      lhs <- 'tu',
                      pref <- pr2
                ),
                pr2 : RSL!PatternRef (
                      name <- 'modify',
                      params <- Sequence{param012, param022, param032,
                                param042}
                ),
                param012 : RSL!RealParameter(
                       name <- 'tu'

                ),
                param022 : RSL!PatternRef(
                       name <- 'search_pattern'
                ),
                param032 : RSL!PatternRef(
                       name <- 'chk_code'

                ),
                param042 : RSL!StringParameter(
                       name <- '' + s.checkptCond.pattern.substring(1,4) +
                              ' '

                )
}
```

APPENDIX D

RSL RULES FOR TANSFORMATIONS

A sample of RSL rule generated by the Rule Generator in FraSPA is provided in Appendix D.1. The generated rules are analyzed and applied by the DMS (PTE that works in the backend) on the existing applications to do the required code instrumentation.

D.1. RSL Rule Generated by the Rule Generator in FraSPA

The following RSL rule (`add_statements`) can be used to add new statements (`add_code1`) after the statement specified as search-pattern (`search_pattern1)` in the following code.

```
default base domain Cpp~VisualCpp6.

pattern add_code1() : statement_seq
  =
  "\>Cpp~VisualCpp6\:[ statement = expression_statement] b =
    \>Cpp~VisualCpp6\:[ postfix_expression = primary_expression]
     \>Cpp~VisualCpp6\:[unqualified_id = template_id]
       exchange<double>
     \<\:unqualified_id
    \<\:postfix_expression
     (b, myrows_Fraspa+2, mycols_Fraspa+2, P_Fraspa, Q_Fraspa,
     p_Fraspa, q_Fraspa, comm2d_Fraspa,rowcomm_Fraspa, colcomm_Fraspa);
  \<\:statement".


pattern search_pattern1() : statement
  =
  "\>Cpp~VisualCpp6\:[ expression = assignment_expression]  b =
    \>Cpp~VisualCpp6\:[ postfix_expression = simple_type_specifier '('
      expression_list ')']
      compute(a, f, b, M, N)
    \<\:postfix_expression
  \<\:expression ; ".


external pattern addCodeAfterStatement(tu : translation_unit, stmt :
statement, s_seq2 : statement_seq, id : IDENTIFIER, id2 : IDENTIFIER) :
translation_unit
  =
'addCodeAfterStatement' in domain Cpp~VisualCpp6.

rule add_statements(
```

```
  tu : translation_unit)
:
  translation_unit->
  translation_unit
=
  tu
->
  addCodeAfterStatement(tu, search_pattern1(), add_code1(), "main ",
"compute ")
  if tu ~= addCodeAfterStatement(tu, search_pattern1(), add_code1(),
"main ", "compute ")
  .

public ruleset r = {add_statements}.
```

APPENDIX E

BACK-END TRANSFORMATION FUNCTIONS

The PARLANSE external functions that are reusable or shared among multiple RSL rules in both Hi-PaL and DALC are shown in Appendix E.1. and E.2. The following PARLANSE external functions are useful for pattern-matching required for applying the RSL rules for transforming the existing applications.

E.1. PARLANSE Function for Searching the Return Statement

The following PARLANSE external function is useful for searching the return statement in the "main" function to insert the MPI_Finalize() library call before it.

```
`this function is used for adding MPI_Finalize() before the return in
`function main.

statement':
(define addFinalize
     (lambda Registry:CreatingPattern
      (value (local (;;
      [representation_instance AST:RepresentationInstance]
      [new_node2 AST:Node]
      [new_node1 AST:Node]
      [search_node AST:Node]
       );;
      (;;
       (= representation_instance
        (AST:GetForestRepresentationInstance
        (AST:GetForst arguments:1 )
        (AST:GetRepresentation arguments:1)))
        (= new_node2 (AST:CreateNode representation_instance
        GrammarConstants:NodeTypes:_statement_seq_2))
        (= new_node1 (AST:CreateNode representation_instance
            GrammarConstants:NodeTypes:_statement_2))
            (AST:ScanTreeNodes  arguments:1
            (lambda (function boolean AST:Node
                    )function
                   (value (local (;; );;
                   (;;
                   (ifthen (== (AST:GetNodeType ?)
                     GrammarConstants:NodeTypes:_jump_statement_3)
                     (ifthen (== (@(AST:GetString ?))
                             (@(AST:GetString arguments:2)))
                      (;;
                            (= search_node (AST:GetParent
                            (AST:GetParent ?)))
                            (ifthen (~= search_node
                             AST:VoidNode)
```

```
                              (;;
                              (AST:ConnectNthChild new_node2 1 arguments:3)
                              (AST:ConnectNthChild new_node2 2 arguments:2)
                              (AST:ConnectNthChild new_node1 1 new_node2)
                             (AST:ReplaceTree search_node new_node1)
                             );;
                          )
                          );;
                        )ifthen
                      )ifthen
                     (return ~t)
                      );;
                     )local
                     ~t
                     )value
                     )lambda
               )
               (return arguments:1)
      );;
      )local
      (void AST:Node)
      )value
      )lambda
)define
```

E.2. PARLANSE Function for Including Helper Files

The following PARLANSE external function is useful for inserting the directives for including helper files in the program that is to be transformed. An example of the file that can be inserted by using the following function is “mpi.h”.

```
(define IncludeFile2
(lambda Registry:CreatingPattern
(value (local (;;
      [representation_instance AST:RepresentationInstance]
      [search_node AST:Node]
      [pp_declaration_seq_node AST:Node]
      [new_node1 AST:Node]
      [func_node AST:Node]
      [id_node AST:Node]
      );;
      (;;
       (= representation_instance (AST:GetForestRepresentationInstance
         (AST:GetForest arguments:1 )  (AST:GetRepresentation
         arguments:1)))
       (= new_node1 (AST:CreateNode representation_instance
           GrammarConstants:NodeTypes: _declaration_seq_2))
       (= search_node (AST:FindChildWithProperty  arguments:1
       (lambda (function boolean AST:Node )function
```

```
          (value (local (;;  );;
               (;;
               (ifthen (== (AST:GetNodeType ?) 867)
                (;;
                 (= func_node (GetChildFromParent ? 1239))

                 (= id_node (GetChildFromParent func_node 1915))

                 (ifthen (== (@(AST:GetString id_node)) `main')
                   (;;
                     (return ~t)
                    );;
                  )ifthen
                );;
               )ifthen
               (return ~f)
               );;
              )local
              ~f
            )value
           )lambda
         )
        )

        (ifthen (~= AST:VoidNode search_node)
        (;;
          (= pp_declaration_seq_node (GetParentFromChild search_node
          855))
          (ifthen (~= AST:VoidNode pp_declaration_seq_node)
               (;;
               (AST:ConnectNthChild new_node1 1 arguments:2)
               (AST:ConnectNthChild new_node1 2 search_node)
               (AST:ReplaceNthChild pp_declaration_seq_node 1 new_node1)
              );;
           )ifthen
         );;
         )ifthen
         (return arguments:1)
       );;
)local
(void AST:Node)
)value
)lambda
)define
```